

Programação prática com Python: Fundamentos

Jorgiano Márcio Bruno Vidal, IFRN/DIATINF

26 de junho de 2022

Sumário

1	Introdução	7
1.1	Conceitos básicos	7
1.2	Linguagem de programação	8
1.3	Programa de computador	9
2	Interpretador Python	11
2.1	O interpretador Python	11
2.2	Tipos e operadores	13
2.3	Variáveis e atribuição	15
2.3.1	Atribuição	16
2.3.2	Uso de variáveis	16
2.3.3	Expressão	17
2.3.4	Reatribuição	17
2.4	comando <code>print()</code>	18
2.5	Atividade	19
2.6	Exercícios	21
3	Primeiro Programa	23
3.1	Programa de computador	23
3.2	Primeiro programa	24
3.2.1	Olá mundo com o editor nano	24
3.3	Entrada e saída de dados	26
3.3.1	Saída	27
3.3.2	Separador	29
3.3.3	Fim de linha	30
3.3.4	Entrada	31
3.3.5	Conversão	33
3.4	Exercícios	34

4	Sistema de tipos	39
4.1	Tipos básicos	39
4.2	Conversão	41
4.3	Um pouco <i>string</i>	42
4.3.1	Tamanho do texto	42
4.3.2	Maiúsculas minúsculas	42
4.3.3	Concatenação	44
4.4	Operações com dados de tipos diferentes	46
4.5	Formatação	48
4.5.1	O método <code>format</code>	48
4.6	Exercícios	50
5	Desvio Condicional	53
5.1	Fluxo de processamento	53
5.2	Desvio de fluxo	54
5.2.1	O <code>else</code> é opcional	55
5.3	Condição	56
5.3.1	Operadores relacionais	57
5.3.2	O comando <code>elif</code>	62
5.3.3	Operadores lógicos	64
5.3.4	Precedência	68
5.4	Recuo de texto (indentação)	70
5.5	Grafo de fluxo de controle	71
5.6	Exercícios	73
6	Listas	87
6.1	Coleções	87
6.1.1	O tipo Python <code>List</code>	88
6.1.2	Acesso aos elementos	88
6.1.3	Leitura de <code>List</code>	90
6.1.4	Escrita de <code>List</code>	91
6.1.5	Adicionar elementos	93
6.2	Principais operações	93
6.2.1	Funções	93
6.2.2	Métodos	95
6.2.3	Referência para <code>List</code>	97
6.2.4	Sublista	99
6.3	<i>String</i> e <code>List</code>	99

6.3.1	Conversão <code>List</code> \leftrightarrow <code>string</code>	100
6.4	Exercícios	100
7	Funções	105
7.1	Introdução	105
7.2	Definição em Python	107
7.2.1	Retorno	108
7.2.2	Parâmetros de função	109
7.3	Funções podem chamar outras funções	112
7.4	Funções recursivas	114
7.4.1	Comportamento recursivo	114
7.4.2	Cálculo do Fatorial	114
7.5	Exemplos de recursividade	116
7.5.1	Exponenciação	117
7.5.2	Divisores de um número	117
7.5.3	Sequencia de Fibonacci	120
7.6	Recursividades em listas	121
7.6.1	Soma dos elementos de uma lista	122
7.6.2	Maior elemento de uma lista de inteiros	123
7.6.3	Como evitar cópias de listas	124
7.7	Exercícios	126

Capítulo 1

Introdução

Este material tem como objetivo ajudá-lo a entender os conceitos essenciais sobre programação de computadores. Antes de começar a escrever programas, vamos entender o que é um computador e porque devemos programá-lo.

1.1 Conceitos básicos

Um **computador** é uma máquina com capacidade de efetuar operações aritméticas de forma rápida e correta. Desde os primeiros computadores construídos, seu principal objetivo é de realizar grandes quantidade de cálculos o mais rápido possível. Com a evolução, redução do tamanho e de preço, associado ao da capacidade de processamento, novas funcionalidades surgiram, como capacidade de processar textos, imagens e sons; mais recentemente, com o surgimento e popularização da Internet, os computadores passaram a ser utilizados no dia-a-dia das pessoas. Internamente, porém, todas estas atividades continuam a ser realizadas mediante operações aritméticas e comunicação com periféricos.

As diferentes possibilidades de uso do computador cresceram tanto quanto a quantidade pessoas que o utilizam. Ferramentas úteis que já existiam, como relógios, agendas, telefones, televisores, rádios e outras tantas de uso diário, passaram a ser construídas contendo um pequeno computador que controla e seu funcionamento.

De forma geral, para que o computador seja usado para um fim específico, é necessário desenvolver um **programa de computador**. Uma característica dos computadores é sua flexibilidade com relação ao que ele pode calcular/processar. Ao se construir um computador é comum lhes conferir a característica de ser **programado**. Dessa forma, um **programa de computador** é uma seqüência de operações que determina o comportamento de um computador. De fato, um computador é uma

máquina que não faz nada! É construída para ser programada, ou seja receber e executar instruções. O resultado da execução das instruções leva a um resultado com algum significado humanamente relevante.

Para que o computador seja programado é necessário conhecer quais instruções o computador é capaz de executar. É nesse contexto que surgem as **linguagens de programação**. Esses idiomas apresentam um conjunto possível instruções que um computador conhece, bem como uma forma de expressá-las, em uma sintaxe e semântica próprias. Em outras palavras, a linguagem de programação define as instruções possíveis e a forma de solicitar que o computador realize tais instruções.

1.2 Linguagem de programação

Uma **linguagem de programação** é uma linguagens específica, artificial, desenvolvida para que seja possível construir programas de computadores. Uma linguagem de programação facilita e acelera o desenvolvimento de um programa de computador.

O computador, inicialmente, reconhece apenas 1 (uma) linguagem: a linguagem de máquina. Esta linguagem é a forma primitiva de programação, porém ela é bastante complexa para os padrões humanos. Com o tempo várias linguagens foram especificadas para facilitar a construção de programas de computador pelos humanos. Dessa forma, para usar uma linguagem que não seja a linguagem de máquina, é necessário configurar o computador para que ele reconheça as instruções da linguagem a ser usada. A configuração é feita através da instalação de aplicativos/ferramentas da linguagem escolhida.

Neste material usaremos a linguagem **Python** para escrever programas de computador. Consideramos que o **Python** esteja corretamente instalado e configurado no computador que você usará. Para instalar consulte as instruções do *site* oficial do **Python**: <http://www.python.org>. A versão usada é a mais recente: **Python 3**.

Da mesma forma que as linguagens naturais de comunicação são usadas para uma pessoa orientar outra na realização de uma tarefa, as linguagens de computadores permitem especificar instruções a serem processadas pelo computador. Existem atualmente diferentes linguagens de programação, assim como nós humanos temos também mais de uma linguagem, ou idioma, para podermos nos comunicar. Nas linguagens naturais de comunicação, como português, alemão, francês, italiano, espanhol, sueco, japonês, chinês e outras tantas, a mesma informação pode ser passada de diferentes formas.

As várias linguagens desenvolvidas para os computadores permitem uma variedade de formas de programar. Algumas linguagens podem ser mais adequadas para programar jogos, por exemplo!!! Outra para fazer cálculos trigonométricos. Tam-

bém podemos ter várias linguagens com o mesmo objetivo. Por exemplo, **Python** e **Ruby** são duas linguagens populares e equivalentes!

É importante observar que o **Python** é apenas um exemplo prático. Entenda o processo de construção de programas de computador e será capaz de escrever programas de computador em diferentes linguagens.

1.3 Programa de computador

Um **Programa de computador** é um conjunto de instruções escritas em uma **Linguagem de programação** e armazenada de uma forma que possa ser executada pelo computador. Nos computadores pessoais modernos um **programa** é armazenado em um arquivo de texto, que pode ser usado como roteiro a ser seguido pelo computador.

Para que o computador possa executar o programa é necessário um conjunto de mecanismos para que o processo de leitura das instruções seja feita e a sua execução ocorra normalmente, usando os recursos disponíveis no computador. O principal recurso do computador é processador, que é onde as instruções são executadas. Um processador é uma máquina capaz de executar um conjunto de instruções pré-definidas de forma extremamente rápida.

Para o estudo inicial da programação com Python será considerado um computador pessoal com teclado e monitor de vídeo. O teclado é o dispositivo de entrada de dados para o computador e o monitor de vídeo o dispositivo de saída de dados. Tanto a construção do programa, como a solicitação de sua execução e eventual dados requeridos para a execução do programa serão feitas com o uso de teclado. A visualização do programa e a exibição dos resultados produzidos pelo programa serão feitas usando o monitor de vídeo.

Além dos recursos de *hardware* mencionados é necessário que o computador possua um S.O. (Sistema Operacional), que permite usar os recursos de *hardware* de forma simplificada e eficiente.

Com todos estes recursos disponíveis é ainda necessário instalar e configurar aplicativos necessários a construção e execução de programas na linguagem escolhida. Para este material é necessário instalar e configurar o **Interpretador Python**, que é o aplicativo que entende as instruções escritas em Python. Para a construção de programas será usado um editor de texto simples, que será detalhado mais adiante.

Capítulo 2

Interpretador Python

Antes de construir, de fato, um programa em Python vamos aprender como escrever instruções para que o computador possa entender e executar a instrução de acordo com o que queremos.

2.1 O interpretador Python

Considerando que o Python está instalado e configurado corretamente, vamos iniciar o **Interpretador Python**, que é um programa que entende as instruções escritas em Python. O interpretador é um programa de linha de comando (modo texto) e deve ser executado a partir de um terminal de linha de comandos. A forma de abrir um terminal de comandos pode variar entre os sistemas operacionais (S.O.), Segue um resumo de como executar o terminal dos principais sistemas operacionais.

Windows 10 Uma forma de abrir o terminal de comandos é clicar no Botão Iniciar do Windows, e no campo de pesquisa digitar `Prompt` ou `cmd`. Uma janela com o terminal de comandos é aberta e você pode começar a escrever os comandos a serem executados.

Ubuntu Linux Ao realizar um *login* na interface gráfica pressione as teclas `Ctrl`+`Alt`+`t` simultaneamente e uma janela com um terminal de comandos será aberta.

macOS Na pasta **Utilitários**, que fica localizada na pasta **Aplicativos**, executa o Aplicativo **Terminal.app**.

Em qualquer dos casos você terá uma janela para interagir com o computador por meio de texto. A janela é similar a mostrada na figura 2.1. No canto superior direito

é possível observar o *command prompt*, que indica que o terminal está esperando o usuário digitar algum comando. Nesse exemplo o *prompt* contém a sequência de caracteres `$>` , mas o que aparece depende do S.O. e das configurações.



Figura 2.1: Janela do terminal de comandos

Para executar o interpretador digite `python3` no terminal de linha de comando do seu computador. O que aparece inicialmente é a *Shell Interativa do Python*. Uma *shell*, que em inglês significa concha, é um programa no qual você escreve instruções para o computador executar. Uma *shell* de Python permite então que você escreva instruções na linguagem Python para o computador executar, como visto da figura 2.1.

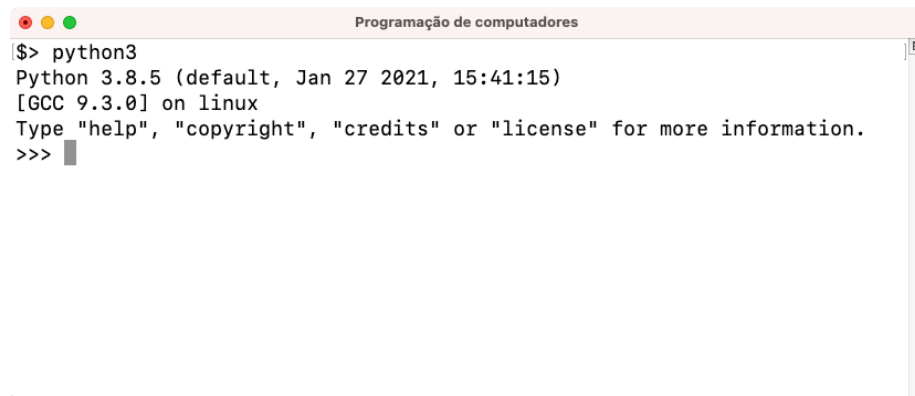
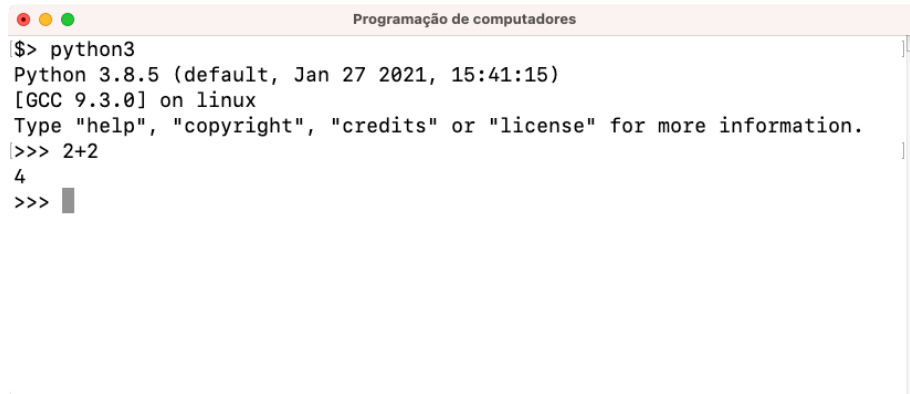


Figura 2.2: Janela do terminal de comandos executando o interpretador Python

Você pode então começar a digitar as instruções na janela. Toda interação com a *shell* é feita usando o teclado. Observe que na linha que contém a sequência de três > (>>>), que é chamado de *prompt*, está um cursor piscando (a barra vertical), este é o ponto onde você começa a escrever as instruções. Em Python, o >>> indica que o interpretador está pronto para receber instruções. Como exemplo vamos pedir para o Python calcular quanto é $2 + 2$. O resultado é mostrado na linha logo abaixo onde você digita, como podemos observar na figura 2.1.



```
$> python3
Python 3.8.5 (default, Jan 27 2021, 15:41:15)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+2
4
>>> █
```

Figura 2.3: Interpretador Python mostrando o resultado da instrução de cálculo $2+2$

Após mostrar o resultado o Python mostra novamente o *prompt* e espera que você digite outra instrução.

Para começar, vamos usar o Python como uma calculadora. Afinal, no início, os computadores foram desenvolvidos para realizar operações matemáticas com maior eficiência do que se nós, humanos, fizéssemos essas contas.

2.2 Tipos e operadores

Para realizar as atividades você deve ter ciência das várias categorias de dados que o **Python** pode manipular. Assim como na matemática os números são categorizados nos conjuntos dos naturais, inteiros, racionais e reais, entre outros, no Python há também várias categorias que se denominam **tipos de dados**.

Para os números, Python oferece apenas dois tipos numéricos básicos: Inteiro e Real, que correspondem às ideias equivalentes na matemática. O que diferencia um número como sendo de uma dos dois tipos é a presença de um ponto . como

parte do número. Dessa forma, `1.5` é um número do tipo de dados Real, enquanto `15` é do tipo Inteiro.

Como os números (Inteiros ou Reais) é possível realizar uma série de operações, assim como na matemática clássica. A tabela a seguir descreve os 6 operadores aritméticos básicos necessários a realização dessas atividades:

Operação	Operador	exemplo	resultado
Adição	<code>+</code>	<code>10 + 3</code>	<code>13</code>
Subtração	<code>-</code>	<code>40 - 8</code>	<code>32</code>
Multiplicação	<code>*</code>	<code>3 * 5</code>	<code>15</code>
Divisão	<code>/</code>	<code>23/5</code>	<code>4.6</code>
Div. Inteira	<code>//</code>	<code>23//5</code>	<code>4</code>
Resto da divisão	<code>%</code>	<code>23%5</code>	<code>3</code>

Você pode digitar as operações do interpretador e o resultado das operações é mostrado em seguida, como podemos observar a seguir:

```

1  >>> 5 + 3
2  8
3  >>> 8 - 3
4  5
5  >>> 3 * 4
6  12
7  >>> 12 / 3
8  4.0
9  >>> 13 // 5
10 2
11 >>> 13 % 5
12 3
13 >>>
```

Observe que o resultado de `12/3`, na linha 7 foi `4.0`, mostrado na linha 8. Isso acontece porque uma divisão entre dois números do tipo de dados inteiro pode não ter como resultado um número inteiro. A fim de evitar a perda da parte fracionária do resultado, **Python** gera um resultado do tipo Real.

Ainda assim, em alguns casos pode ser necessário gerar um resultado contendo apenas a parte inteira da divisão. Nesse caso, para que a divisão de números inteiros gere um resultado também inteiro o operador é o `//`, como se observa no exemplo `13 // 5`, na linha 9. Na divisão padrão usando o operador `/` o resultado seria `2.6`, mas na na divisão inteira operador `//`, o resultado é `2`, conforme mostrado na linha 10.

A operação de resto da divisão, ou módulo, é representada pelo operador `%` e permite determinar o resto de uma divisão de números inteiros. No exemplo da operação 13 dividido por 5 temos que o valor inteiro da divisão é 2, e o resto, que não pode ser dividido, é 3, como pode ser verificado nas linhas 11 e 12.

Como mencionado antes, o computador tem grande capacidade de realizar cálculos. Considere uma multiplicação de números grandes! Quanto tempo você levaria para calcular $123456789 * 987654321$? Experimente escrever isso na *shell python* e veja como o computador pode realizar rapidamente essa conta. Experimente colocar números ainda maiores e observe como efetuar contas aritméticas pode ser rápido com o auxílio de um computador.

Além do operador sendo usado, o tipo do valor resultante depende também dos tipos dos operandos.

- Para operações com valores inteiros o resultado é um valor inteiro, com exceção do operador de divisão `/`.
`10*2` - O valor resultante é o inteiro 20.
`10/2` - O valor resultante é o real 5.0.
- Se pelo menos 1 (um) dos operandos for real, o resultado é um valor real.
Exemplo: `10.0+2` resulta em `12.0` (valor real).
Para o operador `//`, mesmo o valor resultado considerando apenas a parte inteira, o resultado é real. Exemplo: `10.0//3` resulta em 3.0 (valor real) e `10//3` resulta em 3 (valor inteiro).
Isso acontece porque o valor inteiro é convertido em real antes da operação ser realizada.

2.3 Variáveis e atribuição

Agora que você aprendeu a realizar cálculos de forma rápida com a linguagem Python usando o interpretador, você já pode usar o Python para resolver problemas matemáticos com o auxílio do computador. Quando se resolve algum problema é normal usar o resultado de um cálculo como parte de outro cálculo, realizado posteriormente. Para que isso seja possível é necessário guardar o resultado em algum lugar. Em programação são usadas memórias para guardar um determinado valor para uso posterior. A cada uma dessas memórias é dado um nome para que o programador possa lembrar mais facilmente onde guardou cada dado. Embora em um determinado momento uma memória armazene um valor, durante o fluxo do programa, o valor armazenado pode ser substituído por outro; por essa razão essas memórias são denominadas variáveis.

Variáveis São espaços de memória usado para armazenar um determinado valor. Esse espaço recebe o **identificador**, usado para guardar um valor no espaço e também para recuperar o valor guardado no espaço. O identificador é também chamado de **nome** da variável.

2.3.1 Atribuição

Para armazenar valores em uma variável você deve usar operador de **atribuição**, que em Python é o = (símbolo da igualdade em matemática). Para guardar um valor em uma variável com nome (ou identificador) x , deve-se escrever: **x=10**. O valor inteiro 10 é então guardado (ou **atribuído**) a um espaço identificado pelo nome x .

```
>>> x = 10
>>> nota = 60
```

Observe que nada é mostrado na tela, pois o Python entende que você deseja guardar o valor 60 na variável **nota**. Você pode imaginar uma variável como uma caixa onde você guarda um valor e coloca uma etiqueta como um nome. Nesse caso nomeamos a caixa com a etiqueta **nota** e colocamos o valor 60 na caixa. Assim, sempre que precisarmos usar o valor da nota, verificamos na caixa com a etiqueta **nota**, o valor ali guardado é o que precisamos.

2.3.2 Uso de variáveis

Agora que um valor está armazenado em uma variável é possível, a qualquer momento, usar o valor armazenado. O valor é usado a partir do nome da variável. Dessa forma, para multiplicar o valor armazenado na variável **nota** por 5, por exemplo, escrevemos no *prompt Python* a expressão **nota * 5**.

```
>>> nota = 60
>>> nota * 5
300
>>>
```

Quando você armazena um valor em uma variável você pode usar a variável em qualquer lugar onde usaria o valor. Observe a sequência de comandos Python a seguir:

```
>>> velocidade = 80
>>> tempo = 3
>>> distancia = velocidade * tempo
>>>
```


Ao final desta sequência de atribuições você terá três variáveis, onde cada uma guarda um valor:

Identificador	Valor
velocidade	80
tempo	3
distancia	240

A partir de expressões aritméticas e armazenamento de valores em variáveis já é possível usar o Python para resolver alguns problemas.

2.3.3 Expressão

O operador de atribuição, representado pelo símbolo "=", permite que um valor, obtido a partir de uma **expressão** do *lado direito*, seja associado ao espaço de memória identificado pela variável do *lado esquerdo* do operador. A regra de atribuição é

ID = EXPR

Dessa forma, o lado direito contém uma expressão de acordo com as regras da linguagem. As atribuições a seguir são válidas na linguagem Python:

```
a = 10
b = 20
c = a+b
nota1 = 5.4
nota2 = 8.8
media = ((nota1*2)+(nota2*3))/5
v0=0.0
s0=0.0
t=10
a=3.0
s = s0+v0*t+a*t*t/2
r = 5
area = 3.1515*r*r
```

2.3.4 Reatribuição

A separação entre **lado direito** e **lado esquerdo** da atribuição permite que a variável que vai receber um valor, localizada no lado esquerdo, possa ser usada na expressão que gera o valor, localizada no lado direito.

A atribuição realiza primeiramente o cálculo do lado direito do operador `=` e, com o resultado obtido, realizar de fato a atribuição a variável identificada no lado esquerdo do operador `=`. Isto permite que a variável a ser atribuída possa ser usada no lado direito da atribuição, realizando uma **modificação** no valor armazenado. Considere a seguinte sequência:

```
1 x = 10
2 y = x+5
3 x = x+10
4 z = x+5
```

Na linha 3 pode-se observar que o lado direito tem a expressão $x + 10$, que resulta no valor 20. Esse valor é então atribuído a variável x . O processo de atribuição é feito em duas partes:

1. Python calcula o valor do lado direito.
2. Python pega o resultado e atribui a variável no lado esquerdo.

Diz-se que o valor de x foi **modificado** de 10 para 20. A reatribuição é comum em programas de computador e você perceberá que esta propriedade é bastante útil na construção de programas.

Importante: Sempre que se deseja armazenar uma expressão em uma variável, as operações que estão ao lado direito do comando de atribuição `=` são calculadas antes do armazenamento. Assim, `x = 5 * 4` implica inicialmente no cálculo de $5 * 4$, ou seja, 20; é o valor 20 que é armazenado na variável identificada por x .

2.4 comando `print()`

Usado para mostrar um valor ao usuário, normalmente no monitor do computador. Para mostrar o valor armazenado na variável x , escreva: `print(x)` como mostrado a seguir.

```
>>> x = 10
>>> y = 2.5
>>> z=x*y
>>> print(x)
10
>>> print(y)
2.5
```

```
>>> print(z)
25.0
>>>
```

2.5 Atividade

Esta atividade é um guia para fixar o conteúdo visto até aqui. Para realizá-la é necessário usar o **Interpretador Python**. Abra uma janela com um terminal de comandos e execute o interpretador `python3`.

Entenda o resultado de cada uma das expressões a seguir:

1. Escreva a expressão $1 + 2 + 3$ e verifique que o resultado é 6.
2. Escreva a expressão $10 + 5 * 2$ e verifique o resultado. Entenda como este valor é gerado.
3. Escreva a expressão $10 + 100\%8$ e verifique o resultado.
4. Escreva a expressão $10 + 100/8$ e verifique o resultado.
5. Escreva a expressão $10 + 100//8$ e verifique o resultado.
6. Escreva a seguinte sequencia de instruções:

```
x=10
y=20
w=x+y
x=w+1.0
y=x/2
z=x//2
```

Depois use o comando `print()` para cada uma das variáveis e entenda o valor de cada uma delas

7. A expressão a seguir deveria calcular a média aritmética entre 7 e 9.5: $7+9.5/2$. O resultado não é o esperado. Isso acontece porque a divisão é realizada antes de soma. Para resolver este problema deve-se colocar em prioridade a soma. Em **Python3** usa-se os parênteses. A expressão correta é $(7+9.5)/2$. Verifique e confirme.
8. Escreva a sequencia de atribuições a seguir:

```
x=0.1
x=x+0.1
x=x+0.1
x=x+0.1
x=x+0.1
x=x+0.1
x=x+0.1
x=x+0.1
x=x+0.1
x=x+0.1
print(x)
```

Verifique o valor de x mostrado com o comando `print(x)`. Você consegue explicar o resultado?

9. Digite as expressões a seguir e observe o resultado.

- `49*1/49`
- `1/49*49`

Os resultados são os mesmos? devem ser os mesmos? Por que?

2.6 Exercícios

1. Determine o valor e o tipo de x após a seguinte instrução ser executada
`x=10+3*3`
2. Determine o valor e o tipo de x após a seguinte instrução ser executada.
`x=10+18/2`
3. Determine o valor e o tipo de x após a seguinte instrução ser executada.
`x=10+18//2-1`
4. Determine o valor e o tipo de x após a seguinte instrução ser executada.
`x=1+2*3+40//3%5-2+3*5%2`. Qual o tipo do resultado e porque?
5. Determine o valor e o tipo de x após a seguinte instrução ser executada.
`x=1+2*3+40/3%5-2+3//2*5`. Qual o tipo do resultado e porque?
6. Usando o interpretador **Python3**, calcule a soma dos números de 1 a 10, atribua a variável `x` e mostre seu valor.
7. Usando o interpretador **Python3**, calcule a média aritmética entre 7, 7 e 8, 35, atribua a variável `media` e mostre seu valor.
8. Considere duas variáveis n_1 e n_2 , que armazenam 2 notas em curso qualquer. A nota final do curso é calculada através da média ponderada da fórmula:

$$\text{media} = \frac{n_1 \times 2 + n_2 \times 3}{5}$$

com os valores das duas notas calcule, usando o interpretador **Python3**, a nota final do aluno final e atribua o resultado a variável `media_final`. Ao final mostre seu valor.

9. No interpretador **Python3** crie uma variável `x` com um valor inteiro e logo a seguir uma variável `digito` que contém o último dígito (dígito das unidades) da variável `x`. O valor de `x` é desconhecido Exemplo: Se `x` for 73623, `digito` será 3. Use operações matemáticas para determinar o último dígito de um número inteiro.
10. Considere uma variável `seg`, que armazena um tempo t em segundos. Escreva uma sequencia de instruções no interpretador **Python3** que crie 3 variáveis, `h`, `m` e `s`, que contenham, respectivamente o tempo passado convertido em horas (`h`), minutos (`m`) e segundos (`s`).

11. Considere uma variável **x** com um valor inteiro **desconhecido** de 4 dígitos (exemplos: 1234 ou 1029 ou 3893). Usando apenas a variável **x** e expressões matemáticas crie uma variável chamada **x_invertido** que contenha o conteúdo de **x** com os valores invertido

Exemplos:

1234	4321
9182	2891
1029	9201

Capítulo 3

Primeiro Programa

3.1 Programa de computador

Um programa de computador é uma sequência de instruções escritas em uma linguagem de programação. Para executar um programa em Python é necessário ter o **interpretador Python** instalado no computador.

Para a construção de um programa é também necessário que um editor de texto esteja instalado. O editor é usado para escrever as instruções do programa e salvar em um arquivo. Em python, usa-se a extensão `.py` para identificar que o arquivo contém um programa python. Na prática, programas em Python são um modo de facilitar a reprodução de uma série de comandos, pois em vez de digitar uma série de comandos na interface do interpretador, como visto no capítulo anterior, é possível gravar comandos em um arquivo e indicar ao interpretador para ler os comandos diretamente do arquivo, evitando a redigitação.

Qualquer editor de texto pode ser usado para escrever programas. A observação é que o arquivo deve ser gravado sem informações de formatação. Processadores de texto, que são editor que contém informações de formatação de texto além do texto, não podem ser usados. Exemplos de processadores de texto¹ são: Microsoft word, OpenOffice Writer, Apple Pages, entre outros. Alguns editores de texto são desenvolvidos especificamente para desenvolver programas, como Notepad++, Emacs, Jedit, SublimeText, Kate, SciTE, nano, gedit, entre outros.

Além de editores de texto, para facilitar o desenvolvimento de programas pode-se usar ambientes integrados de desenvolvimento (*Integrated Development Environ-*

¹É possível usar um processador de texto, desde que o arquivo seja salvo sem informações de formatação. Mas fique consciente que processadores de texto não possuem funcionalidades adequadas para edição de programas de computador.

ment). Essas ferramentas integram um editor de texto, interpretador, depurador, analisador, controle de versões, gerenciador de projetos, e mais. Assim, após digitar o programa e salvar em um arquivo, basta selecionar um botão na interface do IDE e o interpretador Python é acionado para executar os comandos no arquivo. Caso você tenha conhecimento suficiente e deseja usar IDEs para o desenvolvimento, use-o.

Para este curso o uso de um editor de texto simples é suficiente e simplifica a aprendizagem de programação. No linux os editores mais simples de se começar a usar são o nano, que possui interface texto, e o gedit, que possui interface gráfica. Fique a vontade para usar o editor preferido.

3.2 Primeiro programa

Vamos escrever um programa simples que mostre uma mensagem ao usuário. Este programa, conhecido como **Olá mundo!**, deve escrever na tela o texto `olá mundo!`. Para escrever o primeiro programa execute o editor de texto escolhido e digite o trecho de código a seguir.

Programa 3.1: `ola.py`

```
1 print("Olá mundo!")
```

3.2.1 Olá mundo com o editor nano

Esta seção contém um tutorial de como escrever um programa em sistemas unixes, tais como Linux e macOS, ou no terminal de linha de comandos `bash` disponível no site **Python anywhere** usando o editor de textos **nano**. O **nano** é um editor simples de interface texto, que pode facilmente ser usado em qualquer terminal de linha de comandos. Caso você queira seguir as instruções descritas nesta seção usando o Windows, há a possibilidade de habilitar o **WSL** (*Window Subsystem for Linux*) e instalar uma distribuição do linux no próprio windows. Dessa forma você tem o terminal de linha de comandos e pode realizar as tarefas aqui descritas.

Para escrever o primeiro programa com o **nano**², digite no terminal do Linux `nano`, seguido do nome do arquivo, como mostrado na figura 3.1.

Este comando solicita ao computador a execução do `nano`. Além do nome do editor em si, é informado o nome do arquivo de texto a ser editado (o programa em Python), `ola.py`. Ao executar o editor de textos `nano`, ele ocupa toda a área do terminal, como mostrado na figura 3.2.

²Normalmente uma instalação padrão do Linux contém o `nano`.



Figura 3.1: Comando para executar o nano em um terminal de comandos.

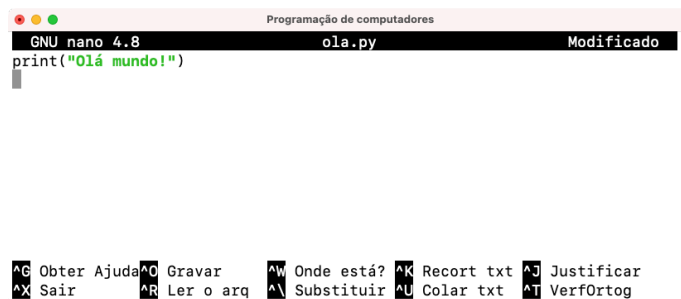


Figura 3.2: Tela do nano

Na parte superior há uma barra com algumas informações sobre o programa em execução, como versão do nano e nome do arquivo editado. Na parte inferior há informações sobre como realizar as principais operações, como Gravar/Salvar, Sair, Ler, etc. Por exemplo, para sair do programa deve-se digitar `Ctrl+X`, como mostrado no canto inferior esquerdo da figura na sequência `^X`. No nano, a tecla `Ctrl` é identificada pelo símbolo `^`.

Todo o restante central da tela do terminal é usada como área de escrita de texto. É nessa área que deve-se escrever o programa. O programa 'Olá mundo!' possui uma única linha, que mostra a mensagem `Olá mundo!` ao usuário. A figura 3.3 ilustra a tela do nano ao editar o arquivo `ola.py`.

Para salvar o arquivo pressione `Ctrl+o` (as teclas `Ctrl` e `o` simultaneamente) e confirme o nome do arquivo. Para sair, pressione `Ctrl+x` e confirme o nome do arquivo a ser salvo, se necessário.

Em terminais unices o comando para listar o conteúdo do diretório atual é o

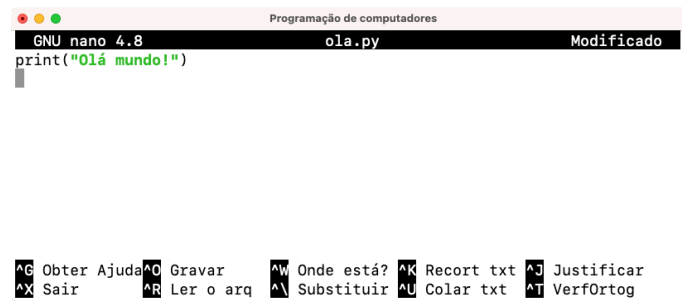


Figura 3.3: Programa Olá mundo digitado no nano.

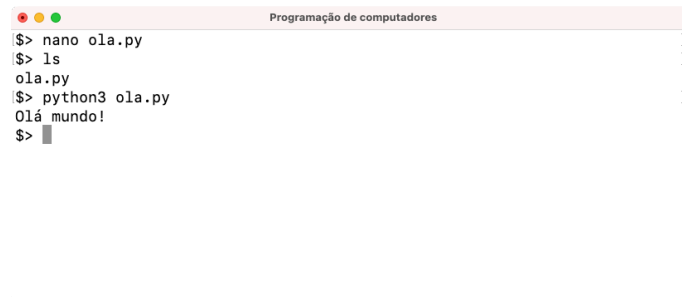


Figura 3.4: Comando para executar o nano em um terminal de comandos.

`ls`, dessa forma é possível listar todos os arquivos do diretório atual³. Depois de digitar `ls` e verificar que o arquivo está salvo, execute o programa digitando `python3 ola_mundo.py` no terminal de comandos, como mostrado na figura 3.4.

Caso o Python esteja instalado e configurado corretamente e o programa esteja correto, a mensagem de texto `Olá mundo!` será exibida na janela do terminal, logo abaixo do comando de execução do programa. Para modificar o arquivo com o programa digite `nano ola.py` no terminal e o nano abre com o texto do arquivo `ola.py`. Fique a vontade para modificar, salvar e executar novamente o programa.

3.3 Entrada e saída de dados

Foi visto até então que por meio de uma linguagem de programação formalizamos o método de dar instruções a um computador, para que o mesmo possa executá-las com precisão.

³Para saber qual é o diretório atual digite o comando `pwd`.

É comum que o programa informe ao usuário o resultado da execução de suas operações. Chama-se **saída** de um programa os valores que ele calcula e mostra. Por padrão um programa executando no terminal de comandos mostra a saída na tela do terminal, em formato texto. O terminal em se que executa o programa é denominado de **saída padrão** do programa. O comando que mostra dados na saída padrão em Python é o `print()`. Mais detalhes sobre o comando `print()` na seção 3.3.1.

Além do programa informar dados, é também comum que o programa necessite de dados para realizar suas operações. O conjunto de dados informados para o programa é chamado de **entrada** do programa. O teclado associado ao terminal em que programa está sendo executado é denominado a **entrada padrão** do programa. O comando para ler dados da entrada padrão é o `input()`, e será detalhado na seção 3.3.4.

Pode parecer estranho mencionar “o teclado associado ao terminal”, pois normalmente um computador possui um só teclado. Isso se deve a razões históricas, em que a um grande computador poderiam estar conectados fisicamente vários pares de tela e teclado; cada um desses pares era denominado de terminal físico. Naquele ambiente, também era possível executar programas a partir de cada um dos diferentes terminais físicos, daí porque o programa tinha a entrada e saída padrão associados à tela e o teclado que formavam um terminal físico onde executado. Atualmente os terminais são virtuais, com uma janela de comandos correspondendo à tela da saída padrão e o único teclado existente sendo compartilhado entre essas várias telas.

3.3.1 Saída

O comando `print()` permite que, durante a execução do programa, mensagens sejam enviadas/**impressas** no terminal para visualização por parte do usuário. Seja o programa `media.py` um programa que calcula a média de aprovação em um curso com 2 notas, n_1 e n_2 . A primeira nota tem peso 2 e a segunda tem peso 3. O programa realiza o cálculo através da fórmula:

$$\text{media} = \frac{n_1 \times 2 + n_2 \times 3}{5}$$

Um exemplo de sequencia de instruções em Python para realizar o cálculo e impressão da média é:

Programa 3.2: `media.py`

```
1 n1 = 9
2 n2 = 7.2
3 media = (n1*2 + n2*3)/5
```

```
4 | print(media)
```

Ao ser executado o programa calcula a média, usando os valores atribuídos às variáveis `n1` e `n2` (linhas 1 e 2), e a atribui à variável `media` (linha 3). A última instrução, na linha 4, **imprime** o valor armazenado na variável `media` para o usuário. Por padrão, a impressão do valor é feita no terminal de comandos.

A função `print()` permite que mais de um valor seja impresso. A lista de valores a serem impressos deve ser separada por vírgula (','). Considere as variáveis `h` e `m`, que armazenam os valores de hora e minuto, respectivamente. Observe `hm.py` a seguir.

Programa 3.3: `hm.py`

```
h = 10
m = 30
print(h,"horas")
print(m,"minutos")
```

Ao ser executado o programa mostra na tela o valor armazenado na variável `h` seguida do texto “horas” em uma linha e o valor armazenado em `m` seguido da palavra “minutos” , em ambos os casos a função `print()` adiciona um separador entre o valor e o texto. Por padrão o separador é o espaço em branco.

```
$> python3 hm.py
10 horas
30 minutos
$>
```

Observe que o texto entre aspas será exibido literalmente (como no caso de “horas” e “minutos”, exibidos como `horas` e `minutos`); por outro lado, um texto sem aspas será substituído por valores, uma vez que representam identificadores de variáveis (`h` e `m` são apresentados na forma dos valores que armazenam 10 e 30, respectivamente).

Agora queremos mostrar a hora em formato `HH:MM`, por exemplo, considerando que o valor das horas e dos minutos estão armazenados nas variáveis `h` e `m`, respectivamente. Com a instrução `print()` padrão isso não é possível, pois `print(h,":",m)` imprime na saída padrão a sequência de 3 valores separadas por espaço entre eles: `10 : 30.`

Esse espaço não é desejado e o que queremos mostrar é: `10:30`. Na próxima seção será mostrado como modificar o separador padrão da função `print()`.

3.3.2 Separador

Considere o programa a seguir, que lê um nome e saúda o usuário com um bom dia.

Programa 3.4: oi.py

```
nome = input()
print("Oi", nome, ", bom dia!")
```

A linha 1 lê ⁴ um dado do usuário e armazena na variável `nome`. A linha 2 mostra uma mensagem de boas vindas personalizada com o nome digitado. O resultado da execução do programa é:

```
$> python3 oi.py
Jorgiano
Oi Jorgiano , bom dia!
$>
```

Observe que a função `print()` insere um espaço em branco entre cada valor impresso, o que não é desejado entre o nome do usuário e a vírgula no texto impresso. No programa `oi.py` a função contém três (3) valores a serem impressos: o texto `"Oi"`, o conteúdo da variável `nome` e o texto `", bom dia"`. Entre cada um desses valores foi adicionado um espaço. É possível especificar o que o comando `print()` deve usar como separador de valores. Usa-se a opção `sep=X`, onde `X` é um texto que especifica o separador de valores da função `print()`. O programa `oi.py`, com a função `print()` especificando a mudança de separador fica:

Programa 3.5: oi.py

```
nome = input()
print("Oi", nome, ", bom dia!", sep="A")
```

o resultado da execução fica:

```
$> python3 oi.py
Jorgiano
OiAJorgiano ,Abom dia!
$>
```

O separador pode ser qualquer texto, inclusive com várias letras. Veja o resultado da execução do programa a seguir:

⁴A função `input()` espera o usuário digitar um texto e será detalhada na seção 3.3.4.

Programa 3.6: oi.py

```
nome = input()
print("Oi", nome, ", bom dia!", sep="meu separador")
```

Agora pode-se modificar o programa `oi.py` para mostrar o texto corretamente:

Programa 3.7: oi.py

```
nome = input()
print("Oi", nome, ", bom dia!", sep="")
```

Ao ser executado se percebe que agora outro problema ocorre: a falta de espaço entre o texto “Oi” e o nome impresso.

```
$> python3 oi.py
Jorgiano
OiJorgiano, bom dia!
$>
```

Para resolver este problema é suficiente colocar um espaço no texto “Oi”. O programa fica:

Programa 3.8: oi.py

```
nome = input()
print("Oi ", nome, ", bom dia!", sep="")
```

Este programa funciona corretamente e mantém a sintaxe como desejado.

Importante: o modificador de separador `sep` deve ser colocado após todos os dados na lista de parâmetros da função `print()`.

3.3.3 Fim de linha

Por padrão, o comando `print()` adiciona uma quebra de linha ao mostrar todos os valores listados como parâmetros. Considere o programa `oi.py` escrito da seguinte forma:

Programa 3.9: oi.py

```
nome = input()
print("Oi")
print(nome)
print(", bom dia!")
```

Uma execução pode ser como mostrado a seguir:

```
$> python3 oi.py
Jorgiano
Oi
Jorgiano
, bom dia!
$>
```

A separação de vários dados a serem impressos em uma mesma linha pode facilitar a compreensão do código escrito, onde a linha a ser impressa contém muitos dados e textos entre os dados. É uma prática comum em programação separar a impressão de uma linha em vários comandos `print()`. Da mesma forma que se modifica o separador de dados, há também como modificar o que será exibido ao fim de cada `print()`, em substituição à quebra de linha. Para tanto, usa-se a opção `end=X` do comando `print()`, como mostrado do código a seguir.

Programa 3.10: oi.py

```
1 nome = input()
2 print("Oi",end=" ")
3 print(nome,end="")
4 print(", bom dia!")
```

Ao terminar a impressão da linha 2 o comando `print()` não adicionar a quebra de linha. Ao invés da quebra de linha, um espaço é adicionado ao mostrar o texto `Oi`. Na linha 3 nada é adicionado ao final da impressão do nome. Finalmente, na linha 4, o texto é impresso e, como não há modificação do final de linha, uma quebra de linha é adicionada, finalizando a linha que começou a ser impressa na linha 2.



É possível perceber que há diferentes formas de se construir um texto de saída de dados de um programa, inclusive com a especificação do separador e do fim de linha em um mesmo comando `print()`: `print(v1,v2,"aquí",x,y,sep=".",end="!")`, desde que eles sejam os últimos parâmetros da função. A escolha de como escrever a saída varia de acordo com as regras de programação do programador/empresa. Fique a vontade para implementar da forma que mais lhe for conveniente.

3.3.4 Entrada

Além de mostrar dados, é comum um programa ler dados a serem informado pelo do usuário. O programa média a seguir, como feito na seção 3.3.1, possui limitações.

```
1 n1 = 9
2 n2 = 7.2
3 media = (n1*2 + n2*3)/5
4 print(media)
```

Para calcular a média de outro aluno é necessário fazer outro programa, ou modificar o programa acima, modificando as atribuições das linhas 1 e 2. É mais salutar construir um programa que seja capaz de receber as notas como dados de entrada, e então calcular a média, usando as notas fornecidas pelo usuário do programa. A função, em Python, para solicitar dados do usuário é a função `input()`.

A função `input()` faz a leitura de dados e retorna sempre um texto (tipo *string*). A linha completa digitada é lida, terminando quando o usuário digita a quebra de linha, atecela **ENTER** ou **RETURN**, dependendo do teclado também pode ser o símbolo  ou . Considere o programa a seguir.

Programa 3.11: prog01.py

```
1 x = input()
2 y = input()
3 print("Valor de x:",x)
4 print("Valor de y:",y)
```

Este programa lê duas linhas e armazena o que foi digitado nas variáveis `x` e `y`, respectivamente. Uma execução do programa pode ser verificada como a seguir.

```
$> nano prog01.py
$> python3 prog01.py
213
Um texto mais completo
Valor de x: 213
Valor de y: Um texto mais completo
$>
```

O conteúdo da variável `x` é `213` e conteúdo da variável `y` é `Um texto mais completo`, no exemplo dado. Ao modificar o programa para ler 2 valores e mostrar a soma, temos o código a seguir.

Programa 3.12: prog01.py

```
1 x = input()
2 y = input()
3 soma = x+y
```



```
4 print(soma)
```

Ao se executar o programa observa-se que o resultado não é o esperado, como mostrado a seguir:

```
$> nano prog01.py
$> python3 prog01.py
10
20
1020
$>
```

O valor mostrado pelo programa é 1020 ao invés do esperado 30. Isso acontece porque o `input()` lê o que foi digitado em forma de texto. O operador `+` para um texto realiza a junção dos mesmos, chamado de concatenação.

3.3.5 Conversão

Como já mencionado, a leitura usando a função `input()` sempre produz um valor do tipo texto/*string*. É necessário usar uma função que **converte** a sequência de símbolos/caracteres de um texto e gera o valor inteiro ou real equivalente. As funções `int()` e `float()` realizam as conversões para inteiro e real, respectivamente.

Modifique o programa anterior para ler os dados, convertê-los em inteiros, realizar a soma, e mostrar o valor da soma. O novo programa é mostrado a seguir.

Programa 3.13: prog01.py

```
1 l = input()
2 x = int(l)
3 l = input()
4 y = int(l)
5 s = x+y
6 print(s)
```

O programa agora mostra o resultado esperado para a soma entre os dois valores digitados.

```
$> python3 prog01.py
10
20
30
$>
```

Observa-se que a variável `t` é usada 2 vezes, apenas para armazenar temporariamente o valor lido quando o programa espera o usuário informar os dados. Como a leitura está sendo feita e o valor usado é diretamente a conversão para inteiro, é possível ler e converter diretamente na linha que realiza a leitura. O código a seguir simplifica a compreensão do programa que lê dois valores e realiza a soma.

Programa 3.14: prog01.py

```
1 x = int(input())
2 y = int(input())
3 s = x+y
4 print(s)
```

O programa modificado para tratamento de números reais é mostrado a seguir:

Programa 3.15: prog01.py

```
1 x = float(input())
2 y = float(input())
3 s = x+y
4 print(s)
```

3.4 Exercícios

1. Escreva um programa que escreva na tela a seguinte mensagem:

Meu primeiro programa!

2. Escreva um programa que escreva na tela o endereço do IFRN da seguinte forma:

Avenida Senador Salgado Filho, 1559,
Tirol, Natal-RN, Brasil
CEP: 59015-000
e-mail: ccs.cnat@ifrn.edu.br
Telefone: 4005-2600

3. Escreva um programa que leia seu nome e mostre a seguinte mensagem:

Oi [NOME], bom dia!

onde [NOME] é o nome digitado pelo usuário.

Exemplo de execução:

```
$> python3 oi.py
Jorgiano
Oi Jorgiano, bom dia!
$>
```

4. Escreva um programa que leia dois números inteiros e mostre a soma dos mesmos.

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
10 20	30

Exemplo de entrada 2	Exemplo de saída 2
123 32	155

5. Escreva um programa que leia dois números reais e mostre a média aritmética dos mesmos.

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
10 20	15.0

Exemplo de entrada 2	Exemplo de saída 2
10.5 81.56	46.03

6. Escreva um programa que leia duas notas e mostre a média obtida a partir das mesmas, de acordo com as regras do IFRN. Para obter a média use a seguinte

fórmula:

$$media = (nota1 \times 2 + nota2 \times 3) \div 5$$

No IFRN as notas são valores inteiros entre 0 e 100

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
81 63	70

Exemplo de entrada 2	Exemplo de saída 2
66 99	85

7. Escreva um programa leia três (3) números e mostre o produto dos mesmos.

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
10 11 12	1320

Exemplo de entrada 2	Exemplo de saída 2
20 1 12	240

8. Escreva um programa que leia a hora de início e de fim de um evento e mostre o tempo do evento no formato HH:MM. A hora de início e fim é dada em minutos desde o início do dia.

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
100 200	1:40

Exemplo de entrada 2	Exemplo de saída 2
500 650	2:30

9. Escreva um programa que leia a quantidade de dias desde o início do ano e mostre quantas semanas e quantos dias já se passaram desde do dia primeiro de janeiro.

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
100	14 semana(s) 2 dia(s)

Exemplo de entrada 2	Exemplo de saída 2
182	26 semana(s) 0 dia(s)

10. Escreva um programa que leia o valor de um item a venda, a quantidade de itens que você vai comprar e o valor que você tem para pagar. Todos os valores são inteiros. O programa deve então informar o valor total a ser pago e o valor do troco que você vai receber.

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
28 3 100	A pagar: 84 Troco : 16

Exemplo de entrada 2	Exemplo de saída 2
65 2 221	A pagar: 130 Troco : 91

11. Escreva um programa que leia a distância entre duas cidades **A** e **B**, em quilômetros, a velocidade, em **km/h**, do carro e mostre qual o tempo da viagem entre

A e **B**, no formato HH:MM. Os segundos devem ser desprezados.

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
280 95	2:56

Exemplo de entrada 2	Exemplo de saída 2
98 52	1:53

12. Escreva um programa que calcule a quantidade de postes a serem colocados em uma rua. O programa deve ler a distância do início ao fim da rua, em quilômetros e a distância entre dois (2) postes, em metros. Seu programa deve mostrar a quantidade de postes e a distância entre os dois últimos postes. Lembre-se que há sempre um poste no início da rua e outro no final. A distância entre cada par de postes é o valor, em metros, lido pelo programa, com exceção da distância entre os dois últimos postes da rua.

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
2 30	68 poste(s). 20 metro(s).

Exemplo de entrada 2	Exemplo de saída 2
5 40	126 poste(s). 0 metro(s).

Exemplo de entrada 3	Exemplo de saída 3
1 550	3 poste(s). 450 metro(s).

Exemplo de entrada 4	Exemplo de saída 4
1 1002	2 poste(s). 1000 metro(s).

Capítulo 4

Sistema de tipos

Em linguagens de programação as regras de manipulação de dados são determinadas pelo sistema de tipos definido. Tipos podem ser definidos como categorias de dados; cada dado manipulado por um programa devem ser enquadrado em uma dessas categorias (ou tipo). Python possui **Tipagem dinâmica**, indicando que o tipo de um dado armazenado em uma variável é definido apenas durante a execução do programa. Podendo, inclusive, ser usada a mesma variável para armazenar valores de diferentes tipos durante a execução do programa. As regras e operações a serem usadas em um determinado dado dependem do **tipo** do dado. Neste capítulo serão explicadas algumas regras e operações a serem realizadas com os tipos de dados básicos de Python.

4.1 Tipos básicos

Python possui quatro tipos de dados básicos: inteiro, real, texto e lógico (conhecido também como *boolean*).

Inteiro é identificado pela palavra reservada `int`. Valores inteiros representam quantidade inteiras, podendo ser escritos sem um componente fracionário (<https://pt.wikipedia.org/wiki/Inteiro>). Exemplos de números inteiros são: `10`, `0`, `-213`, `8392898127388219` e `-293990189231`.

Real é identificado pela palavra reservada `float`. Valores reais representam valores contínuos, que podem ser representados em forma de frações (<https://pt.wikipedia.org/wiki/Fração>). Exemplos de valores reais: `1.5`, `0.1`, `102321.12312`, `10.0`, `-5.1` e `3.333`. Em programação o termo *float* é usado

porque o número é representado com um ponto¹ separando a parte inteira da parte fracionária. O valor `10.0` é identificado como fracionário porque há um ponto, caracterizando um número fracionário, mesmo que a parte fracionária seja `0`. Nesse caso, a quantidade `10` pode ser representada tanto em inteiro como fracionário. A melhor forma depende do problema em questão e deve ser analisada caso a caso. Em computação, a representação de números fracionários e suas operações são definidas pela especificação IEEE-754. Mais detalhes em https://pt.wikipedia.org/wiki/Vírgula_flutuante (do original em inglês https://en.wikipedia.org/wiki/Floating-point_arithmetic).

Texto é uma sequência de símbolos pertencentes a um alfabeto. Em computação um texto é chamado de **cadeia de caracteres** ou **string**. Um valor do tipo **string** é identificado pela palavra reservada `str`. Python reconhece uma *string* quando os símbolos são escritos no código fonte do programa entre aspas. Exemplos de *string* são: `"oi"`, `"Bom dia!"`, `"Olá mundo"`, `"O rato roeu a roupa do rei de Roma."`, `"1"`, `"0"`, `"10.5"`, `"1.5"`, `"-2"`, `"-5.125"` e `"10"`. Observe que a *string* `"10"` não representa a quantidade `10`, mas uma sequência de dois (2) símbolos: o símbolo 1 e o símbolo 0.

Lógico É um valor que representa o estado de uma afirmação/proposição, sendo possível apenas 2 (dois) valores: **Verdadeiro** ou **Falso**. Também chamado de valor *booleano*, pois deriva da definição da **Álgebra de Boole**. Mais detalhes sobre valores *booleanos* serão discutidos no capítulo 5.

Para saber o tipo de um determinado valor usa-se a função `type`. Observe o trecho de código a seguir:

```
1 print("Tipo de 10      :",type(10))
2 print("Tipo de 10.0    :",type(10.0))
3 print("Tipo de 10.0/10 :",type(10.0/10))
4 a = 10
5 b = 1.5
6 c = "0i"
7 d = a+b
8 e = 3
9 f = 10//3
10 g = 10/3
```

¹As linguagens de programação, via de regra, usam o ponto como separador, pois são baseadas na língua inglesa, em contraste com a vírgula da língua portuguesa.


```
11 print("Tipo do valor em a:",type(a))
12 print("Tipo do valor em b:",type(b))
13 print("Tipo do valor em c:",type(c))
14 print("Tipo do valor em d:",type(d))
15 print("Tipo do valor em e:",type(e))
16 print("Tipo do valor em f:",type(f))
17 print("Tipo do valor em g:",type(g))
```

O resultado obtido é:

```
Tipo de 10          : <class 'int'>
Tipo de 10.0        : <class 'float'>
Tipo de 10.0/10     : <class 'float'>
Tipo do valor em a: <class 'int'>
Tipo do valor em b: <class 'float'>
Tipo do valor em c: <class 'str'>
Tipo do valor em d: <class 'float'>
Tipo do valor em e: <class 'int'>
Tipo do valor em f: <class 'int'>
Tipo do valor em g: <class 'float'>
```

A variável *a* possui um valor do tipo inteiro², identificada pela palavra `int`. O tipo do resultado de uma operação depende dos operandos, como veremos no decorrer do capítulo.

4.2 Conversão

Python fornece funções para converter valores entre os tipos básicos. O sucesso da operações de conversão depende do valor e do tipo do dado a ser convertido.

Para converter um valor para inteiro usa-se a função `int(v)`, onde *v* é o valor a ser convertido. Observe o trecho de código a seguir:

```
1 x1 = 242.0
2 x2 = "3"
3 x3 = int(x1)/int(x2)
4 print(x3)
```

²Em programação orientada a objetos - POO - uma **classe** determina um tipo, por isso o Python mostra que um valor possui `<class 'int'>`.

A divisão que ocorre na linha 3 só é possível porque os valores armazenados nas variáveis `x1` e `x2` são convertidos para o tipo inteiro. Caso não fossem convertidos ocorreria um erro na divisão entre um valor do tipo real e um valor do tipo *string*.

Para converter um valor para um número fracionário usa-se a função `float()`. Qualquer valor pode ser convertido para um texto, usando-se a função `str()`.

As funções de conversão `int()` e `float()` já foram usadas na leitura de dados do teclado, pois um valor lido pela função `input()` é sempre do tipo *string*.

4.3 Um pouco *string*

Uma *string* representa sequência de símbolos. Atualmente o uso e manipulação de *strings* permite realizar diferentes tarefas sobre textos em diversos idiomas. Em uma busca na internet é possível digitar uma palavra ou frase e o mecanismo de busca faz uma verificação nas páginas disponíveis publicamente e mostra quais páginas contém a(s) palavra(s) no texto da busca.

Para facilitar a manipulação de textos a linguagem Python, a exemplo da maioria das linguagens de programação, possui uma extensa biblioteca para manipulação de *string*.

4.3.1 Tamanho do texto

Uma operação básica em *string* é a identificação de quantos símbolos ela possui. Para saber quantos símbolos possui a *string* usa-se a função `len(s)`, onde `s` é um valor do tipo *string*, ou uma variável que armazena uma *string*.

4.3.2 Maiúsculas minúsculas

Operações de manipulação de letras são comuns em textos, como trabalhar com letras maiúsculas e minúsculas. Algumas operações estão disponíveis para converter letras entre maiúsculas e minúsculas. Analise o trecho de código a seguir:

Programa 4.1: boas_vindas.py

```
1 print("Qual o seu nome?")
2 nome = input()
3 print("Bom dia",nome)
4 print("Bem vindo ao Python. Qual o seu endereço?")
5 endereco = input()
6 print("Confirmação do endereço:",endereco)
```

Considere a seguintes execução:

```
$> python3 boas\_vindas.py
JOAO
Bom dia JOAO
Bem vindo ao Python. Qual o seu endereço?
RUA DO LADO, 40, CENTRO
Confirmação do endereço: RUA DO LADO, 40, CENTRO
$>
```

O texto anterior, para se adequar as normas da língua portuguesa, necessita de modificações quanto a escrita, especificamente com respeito ao uso de letras maiúsculas e minúsculas. Para tanto, a biblioteca do Python possui métodos ³ para a criação de novas *strings* em que as letras atendem à norma da nossa língua. Nesse sentido, modifique o programa anterior para essa nova versão:

Programa 4.2: boas_vindas.py

```
1 print("Qual o seu nome?")
2 x = input()
3 nome = x.capitalize()
4 print("Bom dia",nome)
5 print("Bem vindo ao Python. Qual o seu endereço?")
6 x = input()
7 endereco = x.lower()
8 print("Confirmação do endereço:",endereco)
```

Na linha 3 é usado o método `capitalize()` de *string* para realizar uma cópia da *string*, em que a primeira letra é convertida em maiúscula e todas as demais letras são convertidas em minúsculas. Evidentemente, se as letras já apresentam a formatação correta, não há necessidade de modificação. Na linha 7 o método `lower()` cria uma nova *string* a partir de `x` modificando todas as letras maiúsculas para minúsculas. Observe um exemplo de execução:

```
$> python3 boas\_vindas.py
Qual o seu nome?
jOSe antONio da silva
Bom dia Jose antonio da silva
```

³Um método é uma operação específica de um uma **classe**. Classes são tipos de dados definidos pelo programador, no contexto do paradigma de programação orientada a objetos. Python é uma linguagem multi-paradigma, permitindo definição de funções e classes.

```
Bem vindo ao Python. Qual o seu endereço?  
RUA DO LADO, 00, cENTro  
Confirmação do endereço: rua do lado, 00, centro  
$>
```

A quantidade de operações definidas para *string* é bastante extensa. A documentação da linguagem Python disponível *online* pode ser consultada em Python-Tipos-Padrão e contém bastante informação sobre as operações dos tipos básicos de Python, incluindo *string*. A seguir enumeramos algumas delas.

4.3.3 Concatenação

A **concatenação** é a justaposição de duas *strings* gerando uma nova, cujo conteúdo é a sequência de todos os caracteres da primeira *string* seguido dos caracteres da segunda. Observe o programa a seguir:

Programa 4.3: concat1.py

```
1 print("Qual o seu primeiro nome?")  
2 nome = input().capitalize()  
3 print("Qual o seu sobrenome?")  
4 sobrenome = input().capitalize()  
5 nome_completo = nome + sobrenome  
6 print("Bem vindo ao Python", nome_completo)
```

A *string* `nome_completo`, definida na linha 5 é formada a partir da concatenação das *strings* `nome` e `sobrenome`. Execute e observe o comportamento do programa. Um exemplo de execução é:

```
$ python3 concat1.py  
Qual o seu primeiro nome?  
jorgiano  
Qual o seu sobrenome?  
vidal  
Bem vindo ao Python JorgianoVidal
```

Para criar a *string* `nome_completo` com o espaço entre o nome e o sobrenome é necessário também concatenar um espaço entre as *strings*. A linha 5 deve então ser:

```
nome_completo = nome + " "+sobrenome
```

A concatenação é feita em 3 *strings*, obtendo o resultado desejado. Na verdade, o que ocorre é similar à operação de adição sobre inteiros, ou seja, quando somamos

três inteiros, os dois primeiros são somados inicialmente e ao resultado é somado o terceiro elemento. Esse raciocínio pode ser aplicado à soma de muitos inteiros e, por equivalência à concatenação de muitos *strings*, desde que os elementos sendo concatenados sejam todos do tipo *string*.

Agora observe o programa a seguir:

Programa 4.4: concat2.py

```
1 a = input()
2 b = input()
3 c = a+b
4 print(c)
```

O programa lê dois valores e mostra a soma dos valores lidos. Ao executar o programa e informar os valores 13 e 12, como mostrado a seguir, o resultado pode parecer estranho para o usuário.

```
$> python3 concat2.py
13
12
1312
$>
```

Afinal de contas a soma entre 13 e 12 deveria resultar em 25. Como já mencionado anteriormente, a função `input()` lê uma sequência de caracteres da entrada padrão e retorna uma *string*. Ao se usar o operador `+` na linha 3 do programa `concat2.py` a operação, de fato, realizada é a concatenação. Isso acontece porque os valores armazenados nas variáveis `a` e `b` são do tipo *string*. Observe a diferença nas duas linhas a seguir:

```
s1 = "13" + "12"
s2 = 13 + 12
```

O valor atribuído à variável `s1` é a concatenação das *strings* “13” e “12”, enquanto o valor atribuído à variável `s2` é a soma dos valores inteiros 13 e 12. Você pode estar se perguntando porque o valor gerado por `input()` é do tipo *string* se aspas não são digitadas. Ocorre que se trata de uma convenção da linguagem Python, como se o interpretador tivesse colocado as aspas antes e depois do que foi digitado. Na verdade, as aspas são usadas dentro dos programas para diferenciar os *strings* dos nomes de variáveis. Na leitura essa diferenciação não é necessária, pois tudo o que é lido já é considerado *string*.

4.4 Operações com dados de tipos diferentes

Operações com dados de tipos diferentes podem ser realizadas, dependendo dos tipos envolvidos e, em alguns casos, dos operadores. Operações aritméticas (+ − × ÷ %) permitem mesclar valores inteiros e reais. Em qualquer operação aritmética envolvendo um valor inteiro e um valor real o resultado será sempre do tipo real. Nesse caso a conversão é feita antes da operação, onde o valor inteiro é automaticamente convertido para real e a operação é feita com dois valores reais. Observe o trecho de código a seguir:

```
1 x = 12
2 y = 8.0
3 resp = x+y
```

A soma dos dois valores armazenados em `x` e `y` é um valor real. O Python, para realizar a operação com dados do mesmo tipo, converte o valor inteiro para real, pois todos os números inteiros possuem um equivalente real. O contrário não é verdadeiro. Considere agora o trecho de código a seguir:

```
1 x = 12
2 y = 8.0
3 z = 3
4 resp = (x+y) // z
5 print(resp)
```

O valor armazenado na variável `resp` é do tipo real, 4.0. A soma de `x` e `y` resulta em um valor real, pois uma conversão implícita é feita no valor armazenado em `x`. A soma, 20.0, é usada na divisão inteira de 20.0 por 3. Nessa operação também há uma conversão implícita do 3 para o equivalente em real 3.0. A operação de divisão **inteira** entre 20.0 e 3.0 resulta no valor 6.0, do tipo **real**, pois o operando são do tipo real e o tipo do resultado é o mesmo dos operandos.

A conversão automática não acontece quando um operando é um valor numérico, inteiro ou real, e o outro é um *string*. Caso deseje realizar operações entre um valor do tipo *string* e um valor numérico é necessário realizar a conversão explícita para uniformizar os tipos. Observe o trecho de código a seguir:

Programa 4.5: prog_erro.py

```
1 x = 12
2 y = "23"
3 resp = x+y
4 print(resp)
```

Na linha 3 há uma tentativa de utilizar o operador `+` com dois operandos de tipos diferentes: inteiro e *string*. Esta operação resulta em erro, como mostrado no exemplo de execução a seguir:

```
$> python3 prog_erro.py
Traceback (most recent call last):
  File "prog_erro.py", line 3, in <module>
    resp = x+y
TypeError: unsupported operand type(s) for +: 'int' and 'str'
$>
```

Observe a mensagem de erro ao se executar o programa. Ela informa que o arquivo `prog_erro.py` contém um erro na linha 3, que é `resp = x+y`. É um erro de tipo (*TypeError*). A descrição do erro, em tradução livre, é “Tipos dos operandos não suportados para `+`: `'int'` e `'str'`”.

Caso deseje realizar esta operação é necessário converter o inteiro para *string* ou a *string* para inteiro. A conversão deve ser explícita porque o resultado difere de acordo com a conversão escolhida. Observe o resultado ao converter *string* para inteiro, como no programa a seguir:

Programa 4.6: `prog_correto1.py`

```
1 x = 12
2 y = "23"
3 resp = x+int(y)
4 print(resp)
```

Observe também o resultado ao converter o inteiro para *string*, como no programa a seguir:

Programa 4.7: `prog_correto2.py`

```
1 x = 12
2 y = "23"
3 resp = str(x)+y
4 print(resp)
```

A escolha de qual conversão fazer deve ser decidida pelo programador dependendo do problema em questão. Você deve entender o problema a ser resolvido e decidir qual conversão é a correta.

4.5 Formatação

Por questão de estética é comum que a forma como o programa mostra os dados tenha que seguir algumas regras. Por exemplo, valores monetários devem sempre ser mostrados com 2 casas decimais. Também, muitas vezes, é conveniente colocar zeros à esquerda de números inteiros. Ao usar o comando `print()` para números reais, por padrão o Python mostra apenas as casas decimais usadas, o que gera falta de padronização na saída. A saída de dados com o comando `print()` é sempre um texto, uma *string*. Quando você usa o `print()` com um valor inteiro ou real, o Python realiza implicitamente a conversão para *string* e mostra no terminal.

4.5.1 O método `format`

Um valor formatado é sempre gerado em forma de *string*. Para mostrar um valor formatado você deve fazer uma conversão usando regras de formatação. A formatação é especificada na forma de um *string* com o formato desejado, o método `format()` e os dados a serem formatados. Observe a linha a seguir:

```
troco_string = "{:.2f}".format(troco)
```

Esta linha formata atribui à variável `troco_string` um texto formatado referente ao valor armazenado na variável `troco`. A regra de formatação é `{:.2f}`, que basicamente indica que o valor deve ser formatado como um número real (f, representa float) com duas casas após o ponto decimal.

Mais precisamente, a sintaxe para formatar valores em Python é

```
"regra".format(valor)
```

onde:

regra contém uma *string* com a especificação da formatação. Deve ser especificada entre chaves. Para números inteiros deve terminar com a letra **d**. Para números reais deve terminar com a letra **f**.

Exemplos:

`"{:5d}".format(x)` Formata o valor armazenado na variável `x` como inteiro. Cria uma *string* com tamanho mínimo 5.

`"{:05d}".format(x)` Formata o valor armazenado na variável `x` como inteiro. Cria uma *string* com tamanho mínimo 5 e preenche com zeros à esquerda se necessário.

"{:5.1f}".format(x) Formata o valor armazenado na variável *x* como real. Cria uma *string* com tamanho mínimo 5 contendo uma casa decimal.

"{:1f}".format(x) Formata o valor armazenado na variável *x* como real. A *string* gerada contém o número real com exatamente 1 casa decimal.

Considere o programa a seguir, que lê um valor a ser pago por um grupo de pessoas e a quantidade de pessoas. O programa deve mostrar qual o valor individual a ser pago.

Programa 4.8: conta.py

```
valor_total = float(input())
qtd_pessoas = int(input())
valor_individual = valor_total/qtd_pessoas
print(valor_individual)
```

Uma conta de 145.50 dividida para 4 pessoa resulta em 36.375. A regra de formatação para valores monetários limita a quantidade de casas decimais a 2. Para adaptar o programa `conta.py` para mostrar o valor de cada um com duas casas decimais você deve modificá-lo para:

Programa 4.9: conta.py

```
valor_total = float(input())
qtd_pessoas = int(input())
valor_individual = valor_total/qtd_pessoas
print("{:.2f}".format(valor_individual))
```

A tabela a seguir resume algumas regras de formatação para números inteiros e reais:

Expressão	Resultado
"{:123d}".format(123)	"123"
"{:5d}".format(123)	"123"
"{:05d}".format(123)	"00123"
"{:+d}".format(123)	"123"
"{:+10d}".format(-12)	-12
"{:f}".format(1.5)	1.500000
"{:20f}".format(1.5)	1.500000
"{:1.2f}".format(1.112)	1.11
"{:10.1f}".format(45.343421321)	45.3
"{:+.1f}".format(200/-3)	-66.7

Para uma descrição detalhada das regras de formatação acesse a página pyformat.info.

4.6 Exercícios

1. Escreva um programa que leia um número inteiro, que corresponde a um sequência de chamada em uma fila de espera, e mostre a mensagem `Proximo: 001`, onde o número deve ser mostra com 3 algarismo, preenchendo com zeros a esquerda.

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
1	Proximo: 001

Exemplo de entrada 2	Exemplo de saída 2
32	Proximo: 032

Exemplo de entrada 3	Exemplo de saída 3
123	Proximo: 123

2. Escreva um programa que calcule o valor de um salário após um reajuste. O programa deve ler da entrada padrão o valor atual do salário e percentual de reajuste como números reais. O program deve calcular o novo valor do salário e mostrar ambos: o valor atual e o novo valor após o reajuste salarial.

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
1000.00 5.5	Atual: 1000.00 Novo : 1055.00

Exemplo de entrada 2	Exemplo de saída 2
1041.27 4.38	Atual: 1041.27 Novo : 1086.88

3. Escreva um programa que leia o raio de um círculo e mostre a área circunferência do mesmo, com exatamente 4 casas decimais.

Considere o valor de $\pi = 3.14159$

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
1	3.1416

Exemplo de entrada 2	Exemplo de saída 2
2	12.5664

Exemplo de entrada 3	Exemplo de saída 3
10	314.1590

Exemplo de entrada 4	Exemplo de saída 4
25	1963.4937

4. Escreva um programa que leia o valor final de venda de um automóvel e calcule seu preço sem impostos, os valores pagos para cada tipo de imposto e imprima os resultados. Considere que, para automóveis populares, o ICMS (Imposto sobre Circulação de Mercadorias e Serviços) é de 18%, o IPI (Imposto sobre Produtos Industrializados) é de 13%, o PIS (Programa de Integração Social) é de 1,4%, e a Cofins (Contribuição para o Financiamento da Seguridade Social) é de 7,6%. Todos os impostos são calculados sobre o valor de custo do automóvel. Todos os valores devem ser mostrados com 2 (duas) casas decimais.

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
24500.0	ICMS: 3150.00 IPI: 2275.00 PIS: 245.00 Cofins: 1330.00 Valor sem impostos: 17500.00

Exemplo de entrada 2	Exemplo de saída 2
63700.0	ICMS: 8190.00 IPI: 5915.00 PIS: 637.00 Cofins: 3458.00 Valor sem impostos: 45500.00

Capítulo 5

Desvio Condicional

A sequência de instruções a ser executada pode depender dos valores da entrada do programa, o que gera uma condição de execução de um trecho de código. Normalmente, ao se processar dados, se faz necessária uma verificação para selecionar/filtrar que dados devem ser usados e que dados devem ser descartados. Além da seleção de dados, uma verificação pode ser feita para determinar quais instruções devem ser executadas a partir de faixas de valores nos dados. Também há a definição do que fazer com o resultado obtido, pois também uma verificação no valor calculado pode determinar diferentes ações baseadas no valor encontrado. Este capítulo introduz a operação de desvio de fluxo de execução, permitindo diferentes fluxos de instruções executadas em um mesmo programa.

5.1 Fluxo de processamento

Um programa de computador executa instruções escritas em uma linguagem de programação. O fluxo de execução determina a **ordem** de execução de cada instrução. Tipicamente, a execução começa na primeira instrução do programa e segue até a última, uma por vez. Instruções de controle de fluxo permitem mudar esse fluxo para que um determinado conjunto de instrução não seja executado ou, de forma simplificada, que a execução de determinada(s) instrução(ões) seja realizada de forma **condicional**. Considere o código do programa a seguir que lê duas notas de um aluno e calcula a média, de acordo com uma regra de ponderação: a primeira avaliação tem peso 2 e a segunda tem peso 3.

```
1 n1 = float(input())
2 n2 = float(input())
3 media = (n1*2+n2*3)/5
```

```
4 print("{:.1f}".format(media))
```

O programa está correto e atende ao que foi solicitado. Porém, com este programa o usuário é que interpreta se o aluno passou por média ou ficou em recuperação. E com uma complicação extra: caso o aluno tenha ficado em recuperação é necessário calcular novamente a média do aluno, agora considerando a nota da recuperação!!!

5.2 Desvio de fluxo

No exemplo em discussão, o próximo passo é fazer o programa determinar se o aluno passou por média ou não. Um aluno passa por média **se** a média do mesmo for igual ou maior do que 6.0. A modificação inicial do programa deve mostrar uma mensagem informando **Aprovado**, caso o aluno tenha atingido a média ou **Em recuperação**, caso o aluno não tenha atingido a média.

$$\text{media} = \frac{2 \times \text{avaliacao}_1 + 3 \times \text{avaliacao}_2}{5}$$

Para determinar em que **condição** o aluno se encontra é necessário realizar uma operação de relacionamento de valores: o conteúdo da variável `media` deve ser igual ou maior do que 6.0. Em python há o comando `if EXPR:` para determinar se uma expressão (`EXPR`) é verdadeira ou falsa. Com o uso do comando `if` pode-se alterar o programa para mostrar apenas a palavra **Aprovado** ou **Em recuperação**, como mostrado no programa a seguir.

```
1 n1 = float(input())
2 n2 = float(input())
3 media = (n1*2+n2*3)/5
4 if media>=6.0:
5     print("Aprovado")
6 else:
7     print("Em recuperação")
8 print("{:.1f}".format(media))
```

Ao se observar a linha 4 percebe-se que há uma condição a ser verificada, `media>=6.0`. A linha 5 está deslocada ligeiramente para a direita, indicando que ela depende da condição da linha 4. Dessa forma a linha 5 só é executada **se** a expressão verificada pelo comando `if` da linha 4 for **VERDADEIRA**. Além do teste da linha 4 há um complemento do comando `if` na linha 6, que é identificado pela palavra reservada `else`. Ao ser feito o teste do comando `if` se o valor avaliado for **FALSO**

os comandos vinculados ao **else**, que no programa mostrado é o que está na linha 7: `print("Em recuperação")`.

Importante O dois pontos (`:`) após a condição no `if` e após o comando `else` é obrigatório em Python.

Dessa forma o fluxo de execução do programa é determinado durante a execução do programa a partir dos valores armazenados em variáveis. No programa, quando a média do aluno é suficiente para a aprovação, apenas **uma linha** é executada entre as linhas 5 e 7.

5.2.1 O `else` é opcional

Considere agora um programa para calcular o salário final de um vendedor em uma loja qualquer. O programa deve ler o salário base do funcionário, o percentual do imposto a ser pago, o valor total de vendas e uma meta de vendas no mês corrente. O imposto é um valor percentual entre 0.0 e 100.0. Com esse dados o programa calcula o salário final de acordo com as regras a seguir:

1. O imposto incide sobre o salário base exclusivamente.
2. O vendedor ganha 5% do total de vendas realizado pelo mesmo.
3. Se o valor de vendas do vendedor for maior do que a meta de vendas, ele ganha um bônus de 3% do valor das vendas que **ultrapassarem** a meta definida.

Com essas regras podemos fazer a parte de leitura de dados do programa

```
salario_base = float(input())
imposto = float(input())
total_vendas = float(input())
meta_vendas = float(input())
```

Com os dados de entrada pode-se calcular o salário final do vendedor, ainda sem o bônus

```
salario = salario_base - (salario_base * imposto / 100.0)
salario = salario + 0.05 * total_vendas
```

O programa está quase pronto. Faltava calcular o bônus que o funcionário deve receber caso tenha ultrapassado a meta de vendas, que é uma condição e modifica o fluxo de execução do programa. Ao salário deve ser adicionado o bônus **se** o vendedor ultrapassar a meta de vendas, mas **nada deve ser feito** caso contrário.

Isso significa que quando a condição a ser colocada no comando `if` for falsa, não há nada a ser colocado no comando `else`. Esse é um caso comum de uso do comando `if` e, por esse motivo, o comando `else` é opcional. A parte de cálculo do bônus pode ser implementada ignorando o comando `else`.

```
if total_vendas > meta:
    salario = salario + (total_vendas-meta)*0.03
```

Após ter realizado todos os cálculos o programa pode mostrar o salário do vendedor formato com duas casas decimais.

```
print("{:.2f}".format(salario))
```

O programa completo de cálculo de salário pode ser visto a seguir.

```
1 salario_base = float(input())
2 imposto = float(input())
3 total_vendas = float(input())
4 meta_vendas = float(input())
5 salario = salario_base - (salario_base * imposto / 100.0)
6 salario = salario + 0.05 * total_vendas
7 if total_vendas > meta:
8     salario = salario + (total_vendas-meta)*0.03
9 print("{:.2f}".format(salario))
```

5.3 Condição

Para que o comando funcione corretamente é necessário que a expressão contida na condição do comando seja do tipo **Lógico**. Tipos lógicos possuem dois valores possíveis: **Verdadeiro** ou **Falso**. Em python valores lógicos possuem o tipo `bool`, que são valores *booleanos*. Mais detalhes podem ser obtidos na definição da **Álgebra de Boole** (https://pt.wikipedia.org/wiki/Álgebra_booleana).

Uma das formas mais comum de se obter um valor lógico é através de operadores que relacionam dois valores, como visto nos exemplos da aprovação e do cálculo do salário. Nestes exemplos foram usados os operadores relacionais `>=` (maior ou igual: \geq) e `>` (maior: $>$).

O valor do tipo python `bool` é reconhecido e pode ser também usado em uma atribuição. O programa de cálculo de aprovação pode ser escrito como mostrado a seguir.


```
1 n1 = float(input())
2 n2 = float(input())
3 media = (n1*2+n2*3)/5
4 aprovado = media>=6.0
5 if aprovado:
6     print("Aprovado")
7 else:
8     print("Em recuperação")
9 print("{:.1f}".format(media))
```

O valor lógico que determina se o aluno foi aprovado é armazenado na variável `aprovado` (linha 4). O valor armazenado na variável `media` é comparado com o valor real `6.0` usando o operador relacional `>=` (maior-igual). O resultado dessa comparação é atribuído a variável `aprovado`, que recebe o valor lógico `True` (VERDADEIRO) ou `False` (FALSO). Pode-se verificar o valor armazenado em `aprovado` através do comando `print` logo após a sua atribuição, como mostrado a seguir.

```
1 n1 = float(input())
2 n2 = float(input())
3 media = (n1*2+n2*3)/5
4 aprovado = media>=6.0
5 print("aprovado =",aprovado)
6 if aprovado:
7     print("Aprovado")
8 else:
9     print("Em recuperação")
10 print("{:.1f}".format(media))
```

OBS: A exibição do valor da variável `aprovado` nesse programa é apenas para efeito de compreensão do valor lógico, verdadeiro ou falso, não sendo adequada para programas reais.

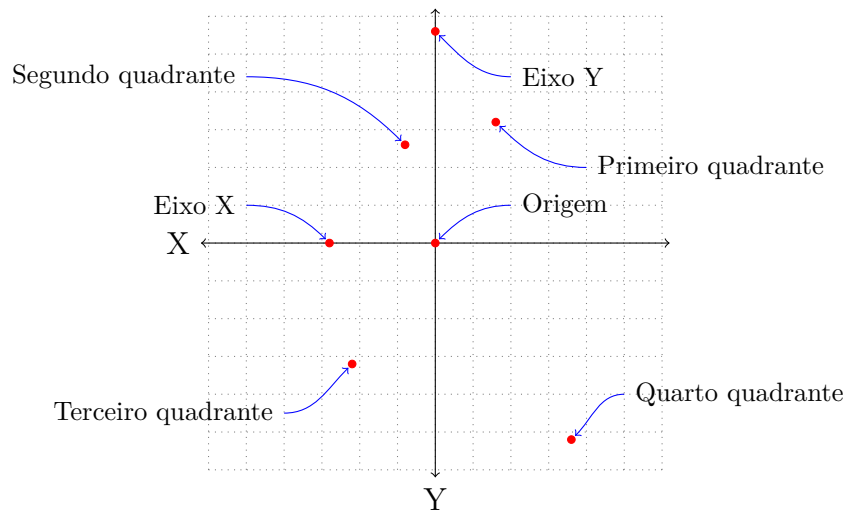
5.3.1 Operadores relacionais

A obtenção de valores lógicos a partir de relação de valores dos tipos básicos de python (inteiro, real e string) pode ser feita usando os operadores mostrados na tabela a seguir.

Operação	Python	exemplo
Igual	<code>==</code>	<code>a==b</code>
Diferente	<code>!=</code>	<code>x!=y</code>
Maior	<code>></code>	<code>conta>em_especie</code>
Menor	<code><</code>	<code>altura<120</code>
Maior igual	<code>>=</code>	<code>media>=6.0</code>
Menor igual	<code><=</code>	<code>quantidade_pessoas<=20</code>

A escolha dos valores e operadores a serem usados na condição do comando `if` depende do entendimento completo do problema a ser resolvido pelo programa. É fundamental entender e saber resolver manualmente o que se está automatizando através da construção do problema.

Exemplo: Problema do quadrante Considere um programa que, dada uma coordenada no plano cartesiano, informe em qual quadrante esta coordenada se localiza, de acordo com a imagem a seguir.



Antes de continuar, tente fazer um programa para resolver este problema. Para teste use as entradas a seguir:

Exemplo de entrada 1	Exemplo de saída 1
3.4 6.321	Primeiro quadrante

Exemplo de entrada 2	Exemplo de saída 2
-5.32 -0.25	Terceiro quadrante

Exemplo de entrada 3	Exemplo de saída 3
-8.5 13	Segundo quadrante

Exemplo de entrada 4	Exemplo de saída 4
12.525 0.0	Eixo X

Exemplo de entrada 5	Exemplo de saída 5
13.13 -17.17	Quarto quadrante

Exemplo de entrada 6	Exemplo de saída 6
0.00 0	Origem

Exemplo de entrada 7	Exemplo de saída 7
0 -17	Eixo Y

Exemplo de entrada 1	Exemplo de saída 1
0.0000000001 -0.00000000025	Quarto quadrante

Você conseguiu implementar corretamente? Bem, uma forma de solucionar esse problema pode ser como descrita a seguir.

O programa deve, primeiramente, ler os valores x e y da coordenada. No plano cartesiano os pontos possuem coordenadas em números reais.

```
x = float(input())
y = float(input())
```

A primeira observação a ser feita é que um ponto localizado na origem, identificada pela coordenada $(0, 0)$ não está em nenhum quadrante. Além da origem, uma coordenada pode estar localizada no eixo X ou no eixo Y , que, assim como a origem, não pertence a nenhum quadrante. Dessa forma, parece ser uma boa abordagem começar verificando se o ponto está localizado na origem, identificada pela coordenada $(0, 0)$.

```
if x==0:
    if y==0:
        print("Origem") # Localizado na origem (x,y) == (0,0)
```

Primeiro o programa verifica se o valor de x é 0 (zero). É necessário fazer mais uma verificação para que coordenada esteja na origem. Dessa forma outro comando `if` é adicionado, **aninhado** com o primeiro.

Comandos aninhados São comandos de controle de fluxo que dependem de comandos de controle de fluxo.

Os dois comandos `ifs` que verificam se uma coordenada está na origem são chamados de **ifs aninhados**.

Com esse trecho de código pode-se observar que um `else` no comando `if y==0:` identifica que a coordenada está no eixo Y , já que o valor de x é 0 (zero) e o de y é diferente de zero. O trecho de código atualizado fica como mostrado a seguir.

```
if x==0:
    if y==0:
        print("Origem") # Localizado na origem (x,y) == (0,0)
    else:
        print("Eixo Y")
```

Com esse trecho de código cobrimos os casos onde a coordenada está na origem ou no eixo Y . Ao colocar um comando `else` no comando `if x==0` é possível cobrir os casos que ainda faltam.

No programa, observe o alinhamento do `else`; ele inicia exatamente na mesma posição que o `if y==0`, indicando que se trata de uma contraposição àquela condição. Se o `else` fosse colocado no alinhamento do comando `if x==0`, ele passaria a representar a alternativa a `x==0`, que nesse caso não é o que se espera. Ainda pior, o Python aceitaria qualquer das duas situações, pois é o programador que deve indicar a qual dos `ifs` o `else` deve se opor, sendo o alinhamento o que caracteriza o par *if-else*.

Em Python esse rigor no alinhamento é tão importante que voltaremos a ele mais adiante.

Considerando a informação sobre alinhamento *if-else* e que no **else** em questão o valor de x será, obrigatoriamente, diferente de 0 (zero) (lembre: *else* significa senão ou caso contrário). Pode-se, então, completar aquele o trecho de código para:

```
if x==0:
    if y==0:
        print("Origem") # Localizado na origem (x,y) == (0,0)
    else:
        print("Eixo Y")
else: # x diferente de 0 (zero)
    if y==0: # eixo X
        print("Eixo X")
    else: # x e Y são diferentes de 0 (zero). coordenada em um dos quadrantes
```

Ao passar pelo comando **else** relativo ao **x==0** é possível verificar se a coordenada está no eixo Y, que ocorre quando a y é 0 (zero). Caso y também não seja 0 (zero) então a coordenada está localizada, obrigatoriamente em um dos quadrantes. Uma forma de realizar a verificação é mostrada no código completo a seguir:

```
1 x = float(input())
2 y = float(input())
3 if x==0:
4     if y==0:
5         print("Origem") # Localizado na origem (x,y) == (0,0)
6     else:
7         print("Eixo Y")
8 else: # x diferente de 0 (zero)
9     if y==0: # eixo X
10        print("Eixo X")
11    else: # x e Y são diferentes de 0 (zero). coordenada em um dos quadrantes
12        if y>0:
13            if x>0:
14                print("Primeiro quadrante")
15            else:
16                print("Segundo quadrante")
17        else: # Nesse caso y é menor do que zero
18            if x<0:
19                print("Terceiro quadrante")
```

```

20     else:
21         print("Quarto quadrante")

```

O programa agora está completo e cobre todos os casos. Existem outras formas de organizar as condições que também estão corretas. Além de diferentes formas aninhar as condições. Fique a vontade para testar diferentes opções.

Ao se observar com mais calma o programa percebe-se que o mesmo realiza a mesma verificação em diferentes pontos. Como exemplo pode-se citar a instrução `if y==0:`, que aparece nas linhas 4 e 9. Este problema pode ser resolvido de outras formas, inclusive com comandos que será visto nas seções seguintes.

5.3.2 O comando `elif`

O comando `elif` permite que um comando `else` que possua em seu corpo apenas um comando `if` possa ser unido em um só comando: `else if`, simplificado para `elif`.

Desconto Considere uma loja em promoção com desconto progressivo, onde o valor comprado indica o percentual de desconto.

1. 10% para valores entre R\$ 100,00 (inclusive) e R\$ 200,00 (exclusivo).
2. 20% para valores entre R\$ 200,00 (inclusive) e R\$ 300,00 (exclusivo).
3. 30% para valores entre R\$ 300,00 (inclusive) e R\$ 400,00 (exclusivo).
4. 40% para valores entre R\$ 400,00 (inclusive) e R\$ 500,00 (exclusivo).
5. 50% para valores a partir de R\$ 500,00 (inclusive).

Escreva um programa que leia o valor total da compra. e mostre o desconto oferecido e o valor final a ser pago.

Antes de verificar a resposta, tente implementar um programa que resolva este problema.

Você conseguiu implementar corretamente?

Para fazer esse cálculo deve-se primeiramente observar como as condições devem ser construídas. Considere o código a seguir:

```

1  total = float(input())
2  if total < 100.0:
3      conta = total
4  if total >= 100.0:

```

```
5   conta = total-0.1*total
6   if total>=200.0:
7       conta = total-0.2*total
8   if total>=300.0:
9       conta = total-0.3*total
10  if total>=400.0:
11      conta = total-0.4*total
12  if total>=500.0:
13      conta = total-0.5*total
14  print("{:.2f}".format(conta))
```

Este programa mostra o valor correto nesse caso. Vamos observar com mais detalhes como este programa funciona. A sequência de execução para o valor 245,00 é descrita a seguir:

1. A linha 1 lê um valor digitado, converte em real e armazena na variável `total`.
2. A condição na linha 4 é avaliada como verdadeira e a linha 5 é executada, atribuindo o valor 220,5 a variável `conta`.
3. A condição na linha 6 é avaliada como verdadeira e a linha 7 é executada, atribuindo o valor 196 a variável `conta`.
4. As condições nas linhas 8, 10 e 12 não são verdadeiras.
5. O comando na linha 12 é executado e o valor da variável `conta` é mostrado.

Ao se analisar a execução é possível perceber que a condição avaliada como verdadeira na linha 2 e o comando executado na linha 3 não tem qualquer influencia no resultado da execução do programa, já que o comando na linha 5 sobrescreve a variável `conta` sem que a mesma tenha sido usada. Para esse programa não há consequências mais graves, pois a execução desnecessária do comando não afeta a corretude do programa. Para perceber que esse comportamento pode ocasionar erro, observe uma implementação alternativa desse programa:

```
1  total = float(input())
2  if (total < 100.00):
3      print("{:.2f}".format(total))
4  if total>=100.0:
5      print("{:.2f}".format(total-0.1*total))
6  if total>=200.0:
```

```

7   print("{:.2f}".format(total-0.2*total))
8   if total>=300.0:
9       print("{:.2f}".format(total-0.3*total))
10  if total>=400.0:
11      print("{:.2f}".format(total-0.4*total))
12  if total>=500.0:
13      print("{:.2f}".format(total-0.5*total))

```

Ao executar o programa e o valor lido for maior do que 500,00 o programa mostra vários valores.

Há mais de uma forma de escrever esse programa de forma correta. Veja a seguir dois programas que calculam o valor com desconto. O programa da esquerda não usa o comando `elif`, mas possui várias vezes a construção `else...if`. O programa da direita substitui o `else...if` por `elif`.

```

1  total = float(input())
2  if total>100.0:
3      conta = total-0.1*total
4  else:
5      if total>200.0:
6          conta = total-0.2*total
7      else:
8          if total>300.0:
9              conta = total-0.3*total
10         else:
11             if total>400.0:
12                 conta = total-0.4*total
13             else:
14                 if total>500.0:
15                     conta = total-0.5*total
16  print("{:.2f}".format(conta))

```

```

1  total = float(input())
2  if total>100.0:
3      conta = total-0.1*total
4  elif total>200.0:
5      conta = total-0.2*total
6  elif total>300.0:
7      conta = total-0.3*total
8  elif total>400.0:
9      conta = total-0.4*total
10 elif total>500.0:
11     conta = total-0.5*total
12 print("{:.2f}".format(conta))

```

Os dois programas mostrados tem exatamente o mesmo comportamento. Em Python, ao substituir a construção `else...if` por `elif` o programa fica mais legível.

5.3.3 Operadores lógicos

O uso dos comandos condicionais `if`, `else` e `elif` é suficiente para permitir o controle preciso de quais instruções executar. Em muitos casos essa combinação de condições

pode levar a trechos de códigos complexos, com muitos `ifs` e `elses`.

Considere o problema a seguir:

Bonus Uma loja está em promoção, em que todos os produtos estão com 10% de desconto. A loja decidiu oferecer um bonus: caso o valor de compra esteja acima de R\$ 500,00 e a quantidade de itens comprados seja maior do que 4 o valor do desconto é 30%. Escreva um programa que leia a quantidade de itens e o valor total da compra (ainda sem aplicação de descontos) e informe o valor a ser pago.

A solução a seguir usa os comandos da forma que foi vista até agora.

```
1 total = float(input())
2 quantidade_itens = int(input())
3 if total >= 500.0:
4     if quantidade_itens > 4:
5         conta = total * 0.7
6     else:
7         conta = total * 0.9
8 else:
9     conta = total * 0.9
10 print("{:.2f}".format(conta))
```

O programa funciona corretamente, mas pode ser melhorado. Observe que o valor do desconto é de 30% quando `total >= 500.0` e `quantidade_itens > 4`. Caso contrário, `else`, o valor do desconto é de 10%.

É aqui que os operadores lógicos simplificam o código. Estes operadores permitem compor expressões lógicas. No problema do desconto tem duas expressões que devem ser verdadeiras para que o desconto seja aplicado. Usa-se a conjunção `e` para identificar que ambas as condições devem ser verdadeiras. Em Python é possível construir expressões lógicas e usar o operador lógico `and` para verificar o valor lógico de duas ou mais expressões. O código do programa de desconto, usando o operador lógico é mostrado a seguir:

```
1 total = float(input())
2 quantidade_itens = int(input())
3 if total >= 500.0 and quantidade_itens > 4:
4     conta = total * 0.7
5 else:
6     conta = total * 0.9
7 print("{:.2f}".format(conta))
```

Além do operador **and** (**e**) é possível usar outros operadores lógicos, conforme mostrado na tabela a seguir.

Operação	Operador	Exemplo
e	and	<code>nota>=2.0 and nota<6.0</code>
ou	or	<code>a==13 or a==10</code>
negação	not	<code>not a==0</code>

Os operadores lógicos realizam as operações sobre valores lógicos: **VERDADEIRO** e **FALSO**. Estes valores, em programação são, normalmente, obtidos a partir de operações relacionais, como visto anteriormente. Um valor lógico é um tipo de dado que pode, também, ser atribuído a uma variável, que, dependendo do problema, pode simplificar o entendimento. O código a seguir é a solução do problema **Bonus**.

```

1 total = float(input())
2 quantidade_itens = int(input())
3 valor_maior = total>=500.0
4 quantidade_e_maior = quantidade_e_itens>4
5 if valor_e_maior and quantidade_e_maior:
6     conta = total * 0.7
7 else:
8     conta = total * 0.9
9 print("{:.2f}".format(conta))

```

Este programa é equivalente ao anterior, porém fica explícito a operação entre dois valores lógicos: `valor_e_maior` (valor é maior) e `quantidade_e_maior` (quantidade é maior). Caso ambos sejam verdadeiros a expressão é verdadeira e o desconto aplicado é de 30%.

Os operadores lógicos esperam que os operandos sejam do tipo lógico¹. Portanto, para um operador lógico binário (como **e**), que trabalha com dois operandos, há quatro possíveis combinações de valores para os operandos. A seguir apresentaremos os resultados esperados dos operadores lógicos para os diversos valores de operandos.

Operador and Permite que a expressão seja verdadeira quando **ambos** os valores lógicos sejam verdadeiros, resultando em falso caso contrário. Dessa forma, entre as

¹Em verdade, Python permite aplicar operadores lógicos a operandos que são de outros tipos, tais como inteiros, reais e strings. Esse suporte, contudo, foge ao escopo desse texto e tem uso em contextos muito específicos.

quatro possíveis combinações de operandos, há um resultado que pode ser um entre dois valores, conforme mostrado na tabela a seguir.

a	b	a and b
V	V	V
V	F	F
F	V	F
F	F	F

Observando a tabela é fácil perceber que o resultado da condição só é verdadeiro quando **ambos** valores, **a** e **b** forem verdadeiros.

Operador or A tabela para este operador também possui 4 (quatro) possíveis combinações. Nesse caso a condição é verdadeira quando qualquer um dos valores for verdadeiro. Dito de outra forma, a condição é falsa em apenas um caso, quando **ambas** forem falsas, como mostrado na tabela a seguir.

a	b	a or b
V	V	V
V	F	V
F	V	V
F	F	F

Operador not A negação realiza uma inversão do valor lógico ao qual a operação é realizada. Há, nesse caso, duas opções: caso seja falso o resultado é verdadeiro; caso verdadeiro o resultado é falso, como mostrado a tabela a seguir

a	not a
V	F
F	V

Álgebra de *Boole*

O estudo das expressões lógicas é feito em um ramo da álgebra chamado álgebra de *Boole*, ou álgebra booliana. O nome é dado em homenagem ao matemático inglês George Boole, que introduziu os elementos do modelo matemático no livro *The Mathematical Analysis of Logic*, de 1847.

O modelo definido e conhecido como álgebra de *Boole* tem sido fundamental para o desenvolvimento da eletrônica digital, consequentemente computadores. Também é provida como base em todas as linguagens de programação moderna. Seu uso não se restringe a eletrônica e computação, sendo também aplicada em teoria dos conjuntos e estatística.

A volta dos quadrantes Refaça o problema de determinar o quadrante usando os conhecimentos adquiridos até agora!!!

A seguir uma solução que funciona corretamente. Veja se a sua solução ficou muito diferente. Tente entender em detalhes o funcionamento dessa solução.

Programa 5.1: quadrante.py

```
1 x = float(input())
2 y = float(input())
3 loc = "Origem"
4 if x!=0 && y==0:
5     loc = "Eixo X"
6 elif x==0 && y!=0:
7     loc = "Eixo Y"
8 elif x>0 && y>0:
9     loc = "Primeiro quadrante"
10 elif x<0 && y>0:
11     loc = "Segundo quadrante"
12 elif x<0 && y<0:
13     loc = "Terceiro quadrante"
14 else:
15     loc = "Quarto quadrante"
16 print(loc)
```

5.3.4 Precedência

Ao se construir expressões com uso dos operadores lógicos, relacionais e aritméticos é fundamental saber como o python realiza as operações, a ordem correta das operações. Por exemplo, a expressão $a+b<d-c$ or $x==y+z*2$ and $w+k<i/2-j\%3$ resulta em um valor lógico. O python avalia uma expressão aplicando os operadores na ordem de precedência definida na tabela a seguir.

Ordem	Operadores	Significado
1	**	Potenciação
2	*, /, //, %	Multiplicação, Divisões real e inteira, Resto
3	+, -	Adição, Subtração
4	==, !=, >, >=, <, <=	Igual, Diferente, Maior, Maior-igual, Menor, Menor-igual
5	not	Negação lógica
6	and	E lógico
7	or	Ou lógico

Operadores de mesma precedência são avaliados da esquerda para a direita, com exceção do operador de potenciação, cuja precedência é da direita para a esquerda. Para entender melhor observe a sequencia de obtida com as expressões a seguir:

Considere a=1, b=5, c=3, x=10, y=8, z=20, w=50

x > c + 5 and y == a + b											
x	>	c	+	5	and	y	==	a	+	b	Substituição de variáveis
10	>	3	+	5	and	8	==	1	+	5	Adição/Subtração
10	>			8	and	8	==			6	Operadores relacionais
	True				and		False				Operador and
					False						

y + c < x and x > y or w < z * y															
y	+	c	<	x	and	x	>	y	or	w	<	z	*	y	Variáveis
8	+	3	<	10	and	10	>	8	or	50	<	20	*	8	Mult/Div/Resto
8	+	3	<	10	and	10	>	8	or	50	<		80		Adi/Sub
	11		<	10	and	10	>	8	or	50	<		80		Relacional
			F		and		T		or		T				and
				F					or		T				or
								T							

A construção das expressões lógicas são fundamentais para a corretude do programa. Através do conhecimento das expressões e como ocorre o fluxo de execução do programa é possível fazer testes para determinar a corretude do programa.

A precedência dos operadores às vezes engana mesmo os programadores experientes. Há uma maneira de explicitar a ordem que operações são executadas, por meio do uso de parênteses. Sempre que parênteses são utilizados, a expressão por eles englobada é executada antes das demais.

Considere, como exemplo o cálculo da média aritmética de dois números, definido

como sendo a metade da soma entre eles. O seguinte programa em Python poderia ser usado para calculá-la.

Programa 5.2: areaTriangulo.py

```
1  print ("Informe os números para cálculo da média")
2  num1 = float(input())
3  num2 = float(input())
4  media = num1 + num2 / 2
5  print ("A média dos números é ", media)
```

Parece natural imaginar que o programa está correto, mas não está. Pela precedência dos operadores, a divisão de `num2 / 2` ocorre antes da soma `num1 + num2`, gerando um resultado distinto do esperado pela operação de média. Você pode facilmente verificar usando os valores 3 e 2 como entrada do programa; o resultado calculado pelo programa e informado ao usuário seria 4, quando o valor correto é 2,5.

Para resolver o problema acima devemos empregar os parênteses, forçando que a adição seja realizada antes da divisão, conforme mostrado o código a seguir.

Programa 5.3: areaTriangulo.py

```
1  print ("Informe os números para cálculo da média")
2  num1 = float(input())
3  num2 = float(input())
4  media = (num1 + num2) / 2
5  print ("A média dos números é ", media)
```

Portanto, use parênteses para redefinir a precedência, mas além disso, sempre que tiver dúvidas quanto a ordem da execução em relação ao que se deseja. Você observará em programas desenvolvidos por outros programadores o uso extensivo dos parênteses, mesmo quando seriam desnecessários. É uma prática que melhora a legibilidade do código e evita resultados inesperados.

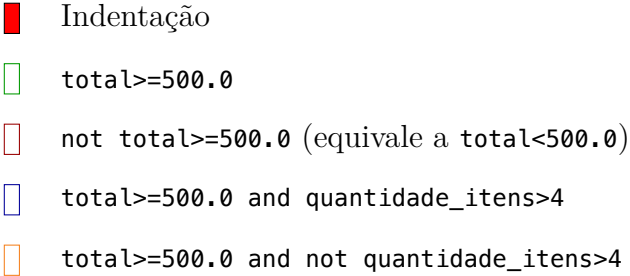
5.4 Recuo de texto (indentação)

Uma boa prática em programação é a de deslocar o texto para a direita, indicando que o comando deslocado é vinculado, ou depende, do comando anterior com um deslocamento menor. Esta prática é chamada de **indentação** do código. No caso de Python a indentação faz parte da sintaxe e semântica da linguagem, é **obrigatória** para que o programa funcione corretamente. Você deve ter percebido que a(s) linha(s)

```

1 total = float(input())
2 quantidade_itens = int(input())
3 if total>=500.0:
4     if quantidade_itens>4:
5         conta = total * 0.7
6     else:
7         conta = total * 0.9
8 else:
9     conta = total * 0.9
10 print("{:.2f}".format(conta))

```



■ Indentação
■ total>=500.0
■ not total>=500.0 (equivalente a total<500.0)
■ total>=500.0 and quantidade_itens>4
■ total>=500.0 and not quantidade_itens>4

Figura 5.1: Espaços em branco são adicionados para definir dependência de instruções

escritas logo após um comando condicional está(ão) "indentada(s)" a direita. Isso indica que a execução está vinculada ao comando `if`.

Em Python, quando usar o comando `if`, todos os comandos que dependem da condição deve estar deslocados a direita com a mesma quantidade de espaços/tabulações. Nos exemplos vistos a indentação usada foi de dois espaços em branco (também é comum o uso de quatro espaços).

A dependência do comando termina quando o Python encontra uma linha com indentação igual ou menor que o comando em questão. Voltando ao primeiro caso do problema do desconto, como mostrado na figura 5.1, as linhas de 4 a 7 são dependentes da condição da linha 3. A linha 5 é dependente da linha 4. A linha 7 da linha 6, que é a continuação do comando da linha 4. A linha 9 é dependente da linha 8 e a linha 10 faz parte do fluxo principal de execução do programa.

Em python é através da indentação que os sub comandos são processados. Não é o caso das maiorias das linguagens. De qualquer forma a semântica de processamento é o que importa.

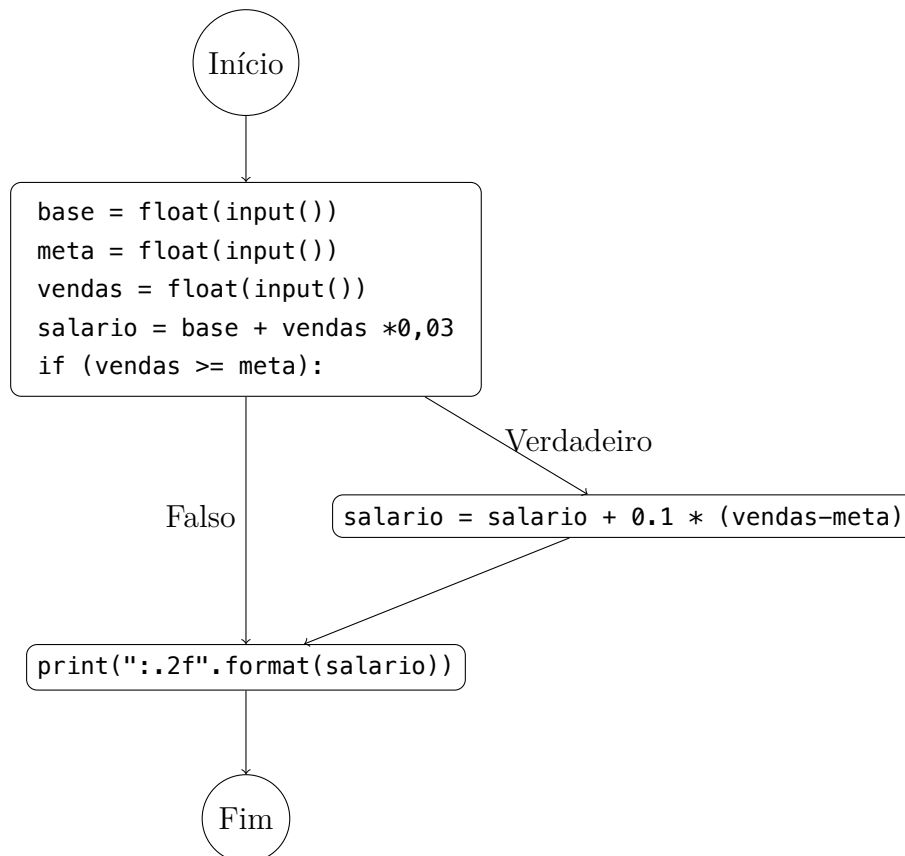
5.5 Grafo de fluxo de controle

Um **CFG** (*Control Flow Graph* - **grafo de fluxo de controle**), também conhecido por *fluxograma*, é uma representação do fluxo de execução de um programa. Através do **CFG** é possível visualizar que partes do programa são executadas e em que condições.

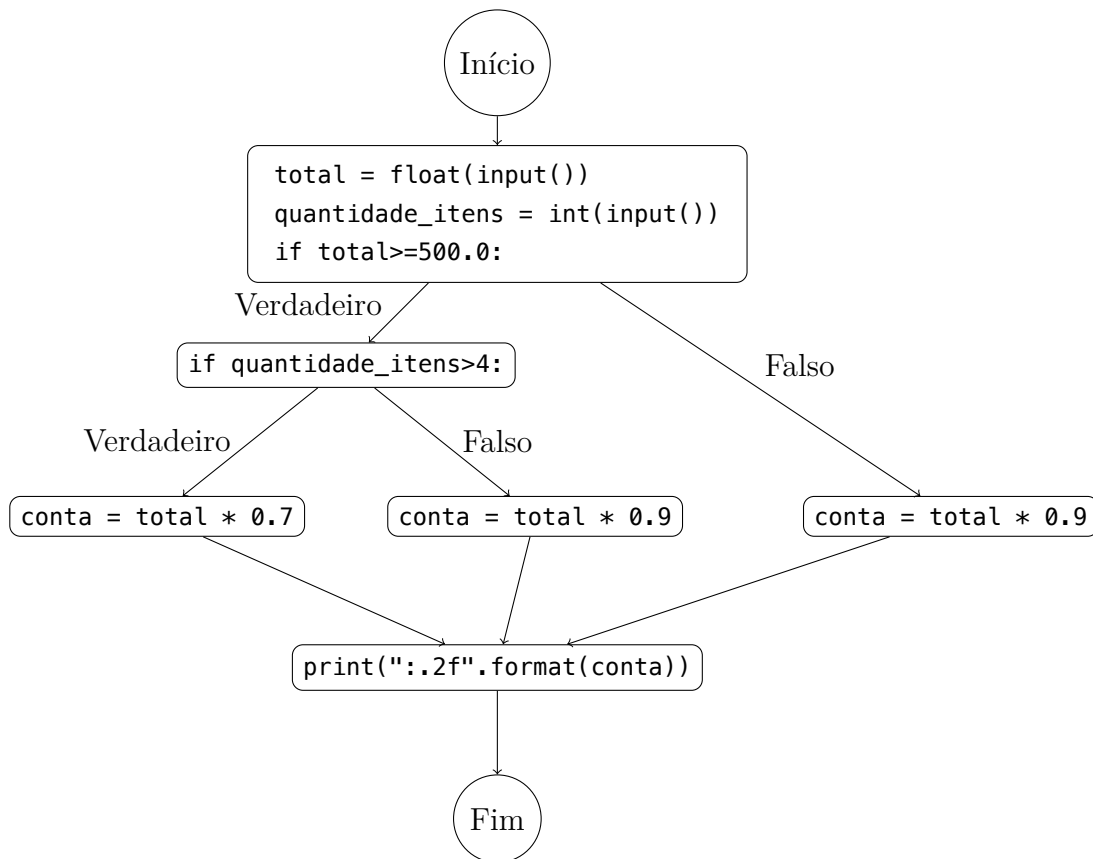
Mesmo em programas simples a construção do **CFG** melhora muito a compreensão do seu funcionamento. Considere o programa a seguir, que determina o salário final de um vendedor.

```
1 base = float(input())
2 meta = float(input())
3 vendas = float(input())
4 salario = base + vendas * 0,03
5 if (vendas >= meta):
6     salario = salario + 0.1 * (vendas - meta)
7 print("{:.2f}".format(salario))
```

O programa lê o salário base, a meta de vendas do mês e o valor total de vendas. O salário final é composto do salário base mais 3% do valor total das vendas. Caso o vendedor tenha passado da meta em valor de vendas, ele recebe um bônus 10% aplicado ao valor que ele passou da meta. O programa mostra o salário final que o vendedor recebe. A seguir o **CFG** do programa de cálculo do salário.



Com relação ao programa do desconto, o **CFG** dá indicações que há o mesmo trecho de código em lugares diferentes do programa. Já foi visto também como eliminar essa duplicidade com o uso de operadores lógicos. O **CFG** ajuda a determinar que trechos de programas são executados para os valores de entrada usados nas expressões lógicas dos comandos condicionais.



5.6 Exercícios

1. Sobre o programa que cálculo de média a seguir.

```
1  n1 = float(input())
2  n2 = float(input())
3  media = (n1*2+n2*3)/5
4  if media >= 6.0:
5      print("Aprovado")
```

```

6  else:
7      print("Em recuperação")
8  print("{:.1f}".format(media))

```

- (a) Desenhe o **CFG** do mesmo.
 - (b) Modifique o programa para que ele mostre uma mensagem de erro caso uma das notas (ou as duas) não possua um valor válido. Um valor válido para a nota deve estar no intervalo $[0.0, 10.0]$ ($0.0 \leq \text{nota} \leq 10.0$).
 - (c) Desenhe o **CFG** do novo programa.
2. Seja o programa, em Python, a seguir (considere que a primeira linha lê três valores inteiros):

```

a,b,c = map(int,input().split())
if (a<b and a<c):
    print("A")
elif (b>c):
    print("B")
else:
    if (c<a or a+b<c):
        print("C")
    else:
        print("D")

```

Escreva a saída do programa, determinando quais linhas são executadas, para as entradas a seguir.

Entrada 1	Saída 1
1 2 3	

Entrada 2	Saída 2
3 2 1	

Entrada 3	Saída 3
2 1 3	

Entrada 4	Saída 4
3 1 2	

- Determine o que o programa da questão anterior mostra se os valores forem iguais.
- Seja o programa, em Python a seguir:

```
a,b,c = map(int,input().split())
x,y,z = map(float,input().split())
if (a*y<b*x and a+b<=c):
    print("A")
elif (a+b<c or x/y<z):
    print("B")
else:
    if (a+c<b and x>y or z>y):
        print("C")
    else:
        print("D")
if (a+b+c>50 and x*y*z>1000):
    print("E")
else:
    print("F")
print("G")
```

Escreva a saída do programa, identificando cada linha executada, para as en-

tradas a seguir.

Entrada 1	Saída 1
1 2 3 3.0 4.1 10.5	

Entrada 2	Saída 2
11 22 33 31.0 41.1 1.5	

Entrada 3	Saída 3
5 12 18 1.6 4.1 10.5	

Entrada 4	Saída 4
5 13 18 1.6 5.5 -1	

Entrada 5	Saída 5
5 13 17 1.6 5.5 -1	

Entrada 6	Saída 6
18 1 17 30.0 2.5 5.0	

Entrada 7	Saída 7
18 1 19 3.0 20.5 -1.5	

5. Com relação a questão anterior, defina valores de entrada para as saídas a seguir:

DICA: Desenhe o CFG para facilitar a visualização do fluxo de execução.

(a) B E G

(b) B F G

(c) D E G

(d) A E G

(e) C E G

6. Desenhe o **CFG** dos dois programas a seguir.

```

1 total = float(input())
2 conta = total
3 if total >= 100.0:
4     conta = conta - 0.1 * total
5 if total >= 200.0:
6     conta = conta - 0.2 * total
7 if total >= 300.0:
8     conta = conta - 0.3 * total
9 if total >= 400.0:
10    conta = conta - 0.4 * total
11 if total >= 500.0:
12    conta = conta - 0.5 * total
13 print("{:.2f}".format(conta))

```

```

1 total = float(input())
2 conta = total
3 if total >= 100.0:
4     conta = conta - 0.1 * total
5 elif total >= 200.0:
6     conta = conta - 0.2 * total
7 elif total >= 300.0:
8     conta = conta - 0.3 * total
9 elif total >= 400.0:
10    conta = conta - 0.4 * total
11 elif total >= 500.0:
12    conta = conta - 0.5 * total
13 print("{:.2f}".format(conta))

```

A partir do **CFG**, faça:

- Escreva o resultado da saída de cada programa para o valor de entrada 50.00.
- Escreva o resultado da saída de cada programa para o valor de entrada 125.50.
- Escreva o resultado da saída de cada programa para o valor de entrada 560.80.

7. Escreva um programa que leia dois números e mostre o maior.

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
1 3	3

Exemplo de entrada 2	Exemplo de saída 2
29103921 932122	29103921

Exemplo de entrada 3	Exemplo de saída 3
13 13	13

Tarefa Desenhe o **CFG** do seu programa e construa casos de teste para cobrir todas as possibilidades (fluxos) identificados no **CFG**.

8. Escreva um programa que leia três números e mostre o menor.

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
1 2 3	1

Exemplo de entrada 2	Exemplo de saída 2
63721 23281 32132	23281

Exemplo de entrada 3	Exemplo de saída 3
5 6 5	5

9. Escreva um programa que leia um número inteiro e informe se o mesmo é par ou ímpar.

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
1	Ímpar

Exemplo de entrada 2	Exemplo de saída 2
13	Ímpar

Exemplo de entrada 3	Exemplo de saída 3
8	Par

Exemplo de entrada 4	Exemplo de saída 4
13749310575	Impar

Exemplo de entrada 5	Exemplo de saída 5
577471044150	Par

10. Escreva um programa que leia 5 números e mostre o maior.

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
1 2 3 4 5	5

Exemplo de entrada 2	Exemplo de saída 2
102 221 133 423 125	423

Exemplo de entrada 3	Exemplo de saída 3
100 200 100 200 100	200

Tarefa Desenhe o **CFG** do seu programa e construa casos de teste para cobrir todas as possibilidades (fluxos) identificados no **CFG**.

11. Escreva um programa que leia 3 números inteiros e informe se a soma dos dois menores é maior do que o maior. o programa deve mostrar a palavra `maior` (toda em minúscula) se a soma dos dois menores for maior do que o maior número. Se a soma dos dois menores for menor que o o maior número o programa deve mostrar a expressão `nao maior`. Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
1 2 3	<code>nao maior</code>

Exemplo de entrada 2	Exemplo de saída 2
4 6 10	<code>maior</code>

Exemplo de entrada 3	Exemplo de saída 3
15 8 10	<code>maior</code>

Exemplo de entrada 4	Exemplo de saída 4
21 31 41	<code>nao maior</code>

12. Escreva um programa que leia 3 números inteiros e verifique se eles formam os lados de um triângulo. Caso seja possível formar um triângulo, o programa deve mostrar a classificação do triângulo com relação aos lados. O programa deve mostrar uma das seguintes opções:

Nenhum caso os números inteiros não representem os lados de um triângulo;

equilatero caso seja um triângulo equilátero;

isosceles caso seja um triângulo isósceles;

escaleno caso seja um triângulo escaleno;

Para formar um triângulo a soma os dois menores lados deve ser maior do que o maior lado.

Exemplo de entrada 1	Exemplo de saída 1
3 5 4	escaleno

Exemplo de entrada 2	Exemplo de saída 2
13 13 13	equilatero

Tarefa Desenhe o **CFG** do seu programa e construa casos de teste para cobrir todas as possibilidades (fluxos) identificados no **CFG**.

13. Escreva um programa que leia uma data e informe se a mesma é válida. A data é informada como 3 números inteiros *dia*, *mes* e *ano*. O programa deve mostrar a palavra *valida* ou *invalida*.
 - Meses com 31 dias: Janeiro, Março, Maio, Julho, Agosto, Outubro, Dezembro.
 - Meses com 30 dias: Abril, Junho, Setembro, Novembro.
 - Mês com 28 dias: Fevereiro. Desconsidere anos bissextos.

Exemplo de entrada 1	Exemplo de saída 1
31 01 2001	valida

Exemplo de entrada 2	Exemplo de saída 2
30 02 2018	invalida

Exemplo de entrada 3	Exemplo de saída 3
07 09 1822	valida

Exemplo de entrada 4	Exemplo de saída 4
31 11 1985	invalida

14. Escreva um programa que leia uma data e informe a data do dia seguinte. Se a data não for válida escrever a mensagem `data invalida`. A data é informada como 3 números inteiros *dia*, *mes* e *ano*. O programa deve mostrar a data no formato `dd/mm/aaaa`.

Exemplo de entrada 1	Exemplo de saída 1
31 01 2001	01/02/2001

Exemplo de entrada 2	Exemplo de saída 2
30 08 1789	31/08/1789

Exemplo de entrada 3	Exemplo de saída 3
31 12 2019	01/01/2020

Exemplo de entrada 4	Exemplo de saída 4
31 04 1912	data invalida

15. **Problema:** Adicionar ímpar ou subtrair par!²

Sejam a e b dois números inteiros.

Em um movimento você pode mudar a da seguinte forma

- Escolher um número inteiro positivo ímpar x , onde $x > 0$, e substituir a por $a + x$;
- Escolher um número inteiro positivo par y , onde $y > 0$, e substituir a por $a - y$.

Escreva um programa que determine a quantidade de operações necessárias para gerar o número b a partir do número a .

Entrada A entrada para o programa é composta de uma única linha com 2 (dois) números inteiros: a e b .

²Adaptado do problema codeforces 1311A: <https://codeforces.com/problemset/problem/1311/A>

Saída A saída é composta de um número inteiro, indicando a quantidade de operações necessárias para obter b a partir de a .

Exemplo de entrada 1	Exemplo de saída 1
2 3	1

Exemplo de entrada 2	Exemplo de saída 2
10 10	0

Exemplo de entrada 3	Exemplo de saída 3
2 4	2

Exemplo de entrada 4	Exemplo de saída 4
7 4	2

Exemplo de entrada 5	Exemplo de saída 5
9 3	1

Nota

No primeiro caso basta adicionar 1.

No segundo caso nada precisa ser feito, então 0 movimentos são necessários.

No terceiro caso é possível adicionar o número 1 duas vezes, realizando dois movimentos.

No quarto caso é possível obter o número 4 ao se subtrair 4 de 7 e somar 1 ao resultado.

No quinto caso apenas um movimento é necessário: subtrair 6 de 9.

16. Problema: Estande de comentaristas!³

Em breve será realizada a copa de futebol para programadores Python. Comentaristas de todo o mundo virão ao evento.

Os organizadores já construíram n estandes para os comentaristas. Um total de m delegações virão cobrir a copa. Toda delegação deve ter a mesma quantidade de estandes. Se algum estande ficar desocupado então as delegações ficarão

³Adaptado do problema codeforces 1186A: <https://codeforces.com/problemset/problem/1186/A>

chateadas, dessa forma **todo** estande deve ser ocupado por exatamente uma delegação.

Se n não for divisível por m é impossível distribuir os estandes para as delegações.

Os organizadores podem construir um novo estande pagando a *burles* e podem demolir um estande pagando b *burles*. Eles podem construir e demolir quantas vezes quiserem, calado pagando o valor de construção ou demolição correspondente. É possível demolir todos os estandes, nesse caso todas as delegações ficam com 0 (zero) estandes.

Escreva um programa que determine o valor mínimo de *burles* que os organizadores devem pagar para satisfazer todas as delegações (fazer com que a quantidade de estandes seja divisível por m).

Entrada A entrada para o programa é composta de uma única linha com 4 (quatro) números inteiros: n , m , a e b , onde n é o número inicial de estandes, m é a quantidade de delegações, a é o valor a ser pago para construir um estande e b o valor a ser pago para demolir um estande.

Saída O programa deve mostrar o valor total em *burles* que os organizadores devem pagar para satisfazer todas as delegações.

Exemplo de entrada 1	Exemplo de saída 1
9 7 3 8	15

Exemplo de entrada 2	Exemplo de saída 2
2 7 3 7	14

Exemplo de entrada 3	Exemplo de saída 3
30 6 17 19	0

Nota:

No primeiro exemplo os organizadores devem construir 5 estandes para fazer um total de 14, pagando 3 *burles* para cada estande construído.

No segundo exemplo os organizadores podem demolir 2 estandes para fazer um

total de 0, pagando 7 burles por cada demolição.

No terceiro exemplo os organizadores já possuem a quantidade exata de estandes para distribuir igualitariamente entre as delegações. Cada delegação fica com 5 estandes.

Capítulo 6

Listas

É comum um programa manipular grande conjuntos de dados, inviabilizando a criação de uma variável para cada dado do conjunto. Também a quantidade de dados a ser processados por um determinado programa pode variar entre uma execução e outra. Neste capítulo será introduzido os elementos de programação para a manipulação de grandes quantidades de dados, sem a definição exata da quantidade.

6.1 Coleções

Coleções são elementos de programação que permitem armazenar vários valores. Uma variável simples armazena um único valor. Quanto mais valores a manipular em um programa, mais variáveis devem ser definidas. Este aumento pode deixar o programa muito complexo, ou mesmo inviável de ser feito.

Considere, a seguir, o programa que lê 5 números e mostra o maior.

```
a,b,c,d,e = map(int,input().split())
maior = a
if (b>maior):
    maior = b
if (c>maior):
    maior = c
if (d>maior):
    maior = d
if (e>maior):
    maior = e
print(maior)
```

Este programa é de fácil entendimento. Um programa para ler 1000 números e mostrar o maior não teria a mesma facilidade de entendimento, além de ser muito grande e ter muitas variáveis. Vamos nos concentrar nas variáveis por enquanto. Como se faz para **armazenar** e **manipular** vários valores em Python? Para manipular é necessário ter formas de armazenar que facilitem a manipulação.

Para armazenar muitos valores as linguagens de programação oferecem diversas estruturas. De forma geral, uma estrutura que armazena muitos valores é chamada de **coleção**. Em Python existem algumas estruturas proveem manipulação de coleções. A mais usada e que será apresentada aqui é o tipo **List**.

6.1.1 O tipo Python List

O tipo **List** permite armazenar, usando apenas uma variável, uma coleção de valores. Com uma variável armazenando muitos valores é necessário uma forma de identificar como acessar cada valor individualmente. Com o tipo **List** os valores são associados com um índice, que é um valor inteiro entre 0 e $n - 1$. Dessa forma cada valor pode ser acessado individualmente usando o índice associado. O código a seguir define um a coleção com 6 números inteiros.

```
nums = [5,2,1,8,3,4]
```

A variável `nums` armazena cada um dos valores e o associa a um índice, como mostrado na figura a seguir.

	0	1	2	3	4	5
nums	5	2	1	8	3	4

Os valores da coleção são todos colocados em sequência (específico de **List**) e cada valor é associado a um índice. O primeiro índice é 0 (zero) e o último é $n - 1$, onde n é a quantidade total de valores armazenados. Nesse exemplo o último elemento tem índice 5, pois há 6 elementos na coleção.

6.1.2 Acesso aos elementos

Para acessar o elemento de uma **List** é necessário informar o índice, além do nome da variável que armazena a **List**. O acesso a um elemento tanto pode ser usado para ler o valor armazenado como para escrever um novo valor a ser armazenado. O índice deve ser escrito entre colchetes logo após o identificador. Observe o programa a seguir.


```

1  nums = [5,2,1,8,3,4]
2  x = nums[2]
3  y = nums[4]
4  nums[1] = 10
5  nums[4] = nums[0] + x
6  z = nums[4]
7  print(x)
8  print(y)
9  print(z)

```

	0	1	2	3	4	5	
nums (antes)	5	2	1	8	3	4	x 1
							y 3
nums (depois)	5	10	1	8	6	4	z 6

Na linha 1 a variável `nums` recebe uma lista com 6 elementos: `[5,2,1,8,3,4]`. Na linha 2 há uma atribuição, onde a variável `x` recebe o valor armazenado no índice 2 de `nums`, que é 1. A linha 3 também tem uma atribuição, com a variável `y` recebendo o valor armazenado no índice 4 de `nums`, que é 3. Na linha 4 um valor é armazenado no índice 1 de `nums`, o valor 10. Em seguida, na linha 5 um outro valor é armazenado na `List`, dessa vez no índice 4. O valor armazenado é obtido pela avaliação da expressão `nums[0] + x`. Na linha 6 é atribuído o valor armazenado no índice 4 de `nums` a variável `z`. As linhas de 7 a 9 mostram os valores das variáveis `x`, `y` e `z`, respectivamente 1, 3 e 6.

O acesso aos elementos de uma `List` funciona normalmente como o acesso a um valor armazenado em uma variável. dessa forma pode ser usado tanto no lado direito como no lado esquerdo de uma atribuição, como visto no exemplo anterior. Também seu uso em expressões é similar ao uso de uma variável. Observe o programa a seguir.

```

1  nums = [5,2,1,8,3,4]
2  maior = nums[0]
3  if (nums[1]>maior):
4      maior = nums[1]
5  if (nums[2]>maior):
6      maior = nums[2]
7  if (nums[3]>maior):
8      maior = nums[3]
9  if (nums[4]>maior):
10     maior = nums[4]
11 if (nums[5]>maior):
12     maior = nums[5]
13 print(maior)

```

```

# maior recebe 5 (índice 0 de nums)
# 2>5 --- FALSO
# 1>5 --- FALSO
# 8>5 --- VERDADEIRO
# Atribuição realizada: maior recebe 8
# 3>8 --- FALSO
# 4>8 --- FALSO
# Escreve na tela 8

```

Na linha 2 o valor armazenado no índice 0 (zero) da `List` `nums` é armazenado também na variável `maior`. Na linha 3 é verificado se o valor armazenado em `nums[1]` é maior do que o valor armazenado na variável `maior`; caso essa condição seja verdadeira, o valor da variável `maior` é atualizado. A mesma operação se repete para todos os valores armazenados na `List` `nums`. Ao final o programa mostra o valor armazenado na variável `maior`, que contém o maior valor entre todos os valores armazenados em `nums`.

6.1.3 Leitura de `List`

Existem várias diferentes formas de se ler dados e colocar em uma `List`. A primeira que será vista é quando todos os dados estão na mesma linha, separados por espaço. O comando `input().split()` realiza a leitura de uma *string* e imediatamente depois separa os valores lidos, gerando uma lista com os vários elementos, usando o espaço como referência de separação. Os espaços são eliminados e não entram na lista.

```
lista = input().split()
print(lista[0])
print(lista[1])
print(lista[5])
```

A quantidade de elementos na lista é igual quantidade de palavras digitadas na linha. É importante lembrar a função `input()` lê a linha como *string* (texto), dessa forma cada elemento da `List` também é um texto. É necessário usar a função `map` para converter cada elemento em inteiro. Considere o programa de mostrar o maior de 6 números armazenando os valores em uma `List`.

```
lista = list(map(int,input().split()))
maior = lista[0]
if (lista[1]>maior):
    maior = lista[1]
if (lista[2]>maior):
    maior = lista[2]
if (lista[3]>maior):
    maior = lista[3]
if (lista[4]>maior):
    maior = lista[4]
if (lista[5]>maior):
    maior = lista[5]
print(maior)
```

Outra forma de fazer o mesmo programa pode ser vista a seguir. Observe que é necessário informar ao Python para gerar uma lista depois de dividir a linha lida da entrada padrão com o comando `input()`.

```
lista = list(map(int,input().split()))
ind_maior = 0
if (lista[1]>lista[ind_maior]):
    ind_maior = 1
if (lista[2]>lista[ind_maior]):
    ind_maior = 2
if (lista[3]>lista[ind_maior]):
    ind_maior = 3
if (lista[4]>lista[ind_maior]):
    ind_maior = 4
if (lista[5]>lista[ind_maior]):
    ind_maior = 5
print(lista[ind_maior])
```

Nesse caso o controle de qual o maior elemento é feito pelo índice da `List`. O uso de índices para manipular os elementos em uma `List` facilita o controle e entendimento do está sendo realizado.

6.1.4 Escrita de `List`

A função `print` funciona normalmente para `List`. Os elementos são mostrados entre colchetes, separados por vírgula. O programa a seguir cria uma lista com 8 elementos e mostra na tela.

```
l = [6,3,8,4,29,13,18,1]
print(l)
```

A ser executado o programa mostra na tela os elementos da lista.

```
$ python3 lista.py
[6, 3, 8, 4, 29, 13, 18, 1]
$
```

Nem sempre o que se quer é a exibição dos valores armazenados em uma `List`, separados por vírgula e entre colchetes. Por isso existem outras formas de se mostrar os elementos de uma `List`.

Desempacotamento de uma List

Uma alternativa para mostrar os valores em uma `List` é o uso de `*` (asterisco) antes do nome da variável, como mostrado a seguir.

```
l = [6,3,8,4,29,13,18,1]
print(*l)
```

Esta opção mostra o conteúdo de `List`, sem os colchetes, separados apenas por espaço. Isso acontece porque o `*` faz um desempacotamento da lista. Ele gera uma sequência de valores a partir dos elementos da lista. Observe o código a seguir.

```
l = [1, 4, 3, 8]
print(*l)
```

Ao fazer a chamada da função `print()` com uma `List` precedida de um asterisco, o Python gera uma sequência de parâmetros para a função. O trecho de código a seguir é equivalente ao anterior.

```
l = [1, 4, 3, 8]
print(l[0],l[1],l[2],l[3])
```

Como a função `print()` pode receber uma quantidade indeterminada de parâmetros, a `List` é **desempacotada** e cada elemento é um parâmetro. As opções da função `sep` e `end` funcionam normalmente.

```
l = [1, 4, 3, 8]
print(*l, sep=' - ',end='')
```

O uso de desempacotamento permite maior flexibilização na impressão de uma `List` na tela. O trecho de código a seguir mostra os elementos de duas formas diferentes.

```
l = [1, 4, 3, 8]

# Método 1
print("{",end='')
print(*l, sep=', ',end='')
print("}")
```

```
# Método 2
print(*l,sep="\n")
```

No primeiro caso os elementos são mostrados entre chaves e separados por vírgula. No segundo cada elemento é mostrado em uma linha.

6.1.5 Adicionar elementos

Uma `List` é uma coleção dinâmica. Em coleções dinâmicas é permitido alterar a quantidade de elementos, inserindo novos elementos ou removendo elementos da coleção.

Para inserir um elemento no final da lista usa-se o método `append(elemento)`. O código a seguir cria uma `List` vazia, com zero elementos e adiciona 3 elementos lidos da entrada padrão, cada um em uma linha. Ao final mostra os elementos lidos em ordem inversa, do último ao primeiro.

```
l = []
x = int(input())
l.append(x)
x = int(input())
l.append(x)
x = int(input())
l.append(x)
print(l[2])
print(l[1])
print(l[0])
```

O método `append()` é muito útil quando é necessário criar uma coleção e adicionar uma quantidade de elementos indefinida, um a um. Mais detalhes serão vistos no capítulo sobre estruturas de controle de fluxo repetitivas (laços), usando do comando `while`.

6.2 Principais operações

A biblioteca padrão do Python oferece algumas operações e métodos para facilitar o uso de `List`. Algumas operações são bastante usadas e, por isso, possuem implementações na biblioteca padrão do Python. A seguir enumeramos algumas dessas operações.

6.2.1 Funções

Funções são trechos de códigos que executam quando são chamados em outras parte do programa. Funções podem receber parâmetros, que são valores passados para que a função realize as operações sobre estes valores. As funções a seguir recebem `List` como parâmetros, realizam uma operação específica e retornam o resultado da operação.

len() Permite obter a quantidade de elementos armazenados na coleção. O trecho de código a seguir mostra a quantidade de elementos lidos em uma *List*.

```
l = input().split()
qtd = len(l)
print(qtd)
```

max() Retorna o maior elemento da *List*. É necessário que os elementos sejam do mesmo tipo e possam ser comparados. Números inteiros e reais satisfazem essa requisição. *Strings* também podem ser comparadas, sendo a menor determinada pela ordem da letra no alfabeto. O trecho de código a seguir lê uma *string*, divide a mesma em palavras e mostra a “maior” palavra.

```
linha = input()
palavras = linha.split()
maior_palavra = max(palavras)
print(maior_palavra)
```

min() Retorna o menor elemento da lista. Observe o programa a seguir e verifique o valor exibido.

```
numeros = [1,-1,2,-2,3,-4,5,-2,10,-25,30]
menor = min(numeros)
print(menor)
```

sum() Retorna a soma dos valores armazenados na coleção. É necessário que os valores possam ser somados entre si. Considere um programa que lê uma quantidade n de números inteiros, digitados na mesma linha, e mostre a média aritmética dos mesmos.

```
numeros = list(map(int,input().split()))
soma = sum(numeros)
quantidade = len(numeros)
media = soma/quantidade
print(media)
```

Ou, de forma mais simplificada, o programa a seguir é equivalente ao anterior.

```
numeros = list(map(int,input().split()))
print(sum(numeros)/len(numeros))
```

sorted() Retorna uma nova lista com os mesmos elementos da lista passada como parâmetro, mas em ordem crescente. O código a seguir lê uma lista de números e gera uma nova lista com números lidos, ordenados não decrescentemente.

```
nums = list(map(int,input().split()))
nums_ord = sorted(nums)
print("Números          :",*nums)
print("Números ordenados :",*nums_ord)
```

6.2.2 Métodos

Um **Método** é uma operação a ser aplicada diretamente no objeto, definida como parte integrante do elemento. Um método acessa diretamente os valores do elemento onde é aplicada. Por exemplo, para adicionar um elemento a uma **List** usa-se o método **append()**, que conhece a estrutura interna da **List** e organiza os valores de forma otimizada. Em Python o nome do método é sempre colocado **após** o nome da variável. Entre o nome da variável e o nome do método coloca-se o ponto (.) para identificar o método a ser usado.

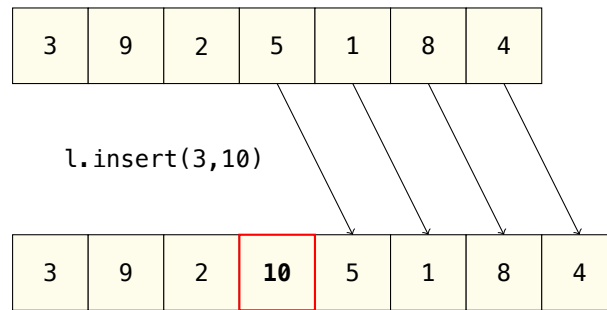
append() Adiciona um novo elemento a **List**. Após essa operação a lista possui um elemento a mais. O novo elemento é inserido no final da lista.

3	9	2	5	1	8	4
---	---	---	---	---	---	---

```
l.append(10)
```

3	9	2	5	1	8	4	10
---	---	---	---	---	---	---	----

insert() Insere um novo valor em um índice específico. Necessário passar como parâmetro o valor a ser inserido e o índice a ser inserido. Caso o índice informado seja menor do que a quantidade de elementos totais na **List** os elementos a partir do índice são deslocados para o próximo índice, como pode ser verificado na figura a seguir. É inserido no índice 3 o elemento 10.



count() Retorna a quantidade de vezes o valor passado como parâmetro existe na `List`. O código a seguir mostra a quantidade de 0 (zeros) na lista digitada pelo usuário.

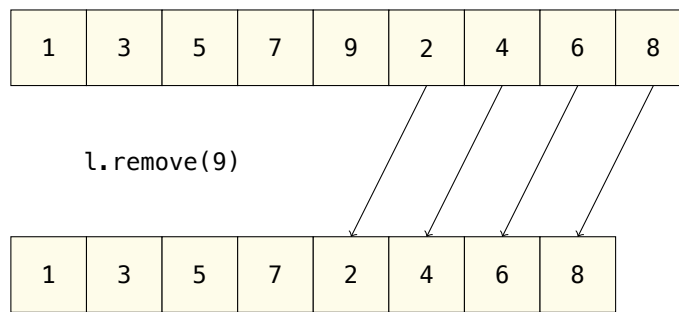
```
numeros = list(map(int,input().split()))
quantidade_de_zeros = numeros.count(0)
print(quantidade_de_zeros)
```

index() Retorna o índice em que um valor está armazenado. Caso haja o mesmo valor em mais de um índice (lugar na `List`) retorna a primeira ocorrência. Se o valor não estiver na `List`, um erro ocorre e o programa para se o erro não for tratado. O programa a seguir mostra o maior elemento e em qual índice ele se encontra.

```
lista = list(map(int,input().split()))
maior = max(lista)
ind_maior = lista.index(maior)
print("O maior elemento é o {:d} e está no índice {:d}".format(maior,ind_maior))
```

remove() Apaga da lista o valor informado como parâmetro. Desloca todos os valores com índice maior do que o elemento removido para o índice menor. A `List` fica com um elemento a menos. Ocorre erro se o valor passado como parâmetro não estiver na lista.

Considere a lista `l = [1,3,5,7,9,2,4,6,8]` e a execução da remoção do elemento 9 com o comando `l.remove(9)`. A lista `l` tem um elemento a menos e o elemento 2 ocupa o lugar do 9. o 4 ocupa o lugar do 2 e assim até o último elemento ocupar o lugar do penúltimo. O lugar do último elemento deixa de existir, como pode ser visto na figura a seguir.



clear() Apaga todos os valores armazenados na `List`. Após a execução do método `clear()` a quantidade de elementos é 0 (zero).

```
l = list(map(int,input().split()))
...
l.clear() # remove todos os elementos da lista.
```

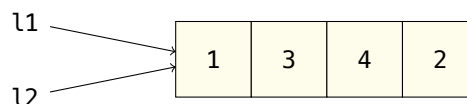
sort() Ordena os elemento da `List`. Após a execução do método `sort()` a quantidade de elementos é exatamente a mesma. Diferentemente da função `sorted()`, o método não cria uma nova `List`. A ordenação é feita trocando os elementos da própria lista de índice entre si.

reverse() Altera a ordem dos elementos na `List`, deixando os elemento em ordem inversa. O primeiro troca de lugar com último, o segundo troca de lugar com o penúltimo, o terceiro com o antepenúltimo, assim sucessivamente até que toda a lista esteja em ordem reversa ao seu estado original.

6.2.3 Referência para List

Quando uma variável é usada para armazenar uma `List`, diferentemente de um tipo primeiro numérico, como inteiro e real, a variável guarda uma **referência** para a `List`. Este fato tem algumas consequências que devem ser conhecidas pelo desenvolvedor.

A primeira coisa a ser bem compreendida é que duas variáveis distintas podem fazer referências a mesma `List`, como mostra a figura a seguir.



As variáveis `l1` e `l2` são **referências** para a mesma `List`. Um dos efeitos deste comportamento é que qualquer alteração feita na `List` usando a variável `l1` tem efeito também na variável `l2`, pois ambas se referem à mesma coleção de dados. Observe o trecho de código a seguir, que usa `l1` e `l2` de acordo com a imagem anterior.

```
1 l1 = [1, 3, 6, 4]
2 l2 = l1
3 l1[1]=2
4 print(l2)
```

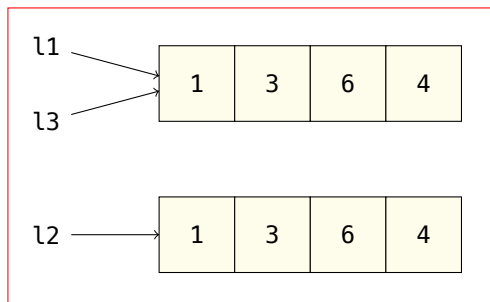
Ao ser executado, observa-se que o valor exibido pela linha 4 é `[1, 2, 6, 4]`. No código não há qualquer modificação usando a variável `l2`, que recebeu na linha 2 a `List` `[1, 3, 6, 4]`.

Cópia Para realizar a cópia de uma `List` usa-se o método `copy()`. Execute e observe o resultado do programa a seguir.

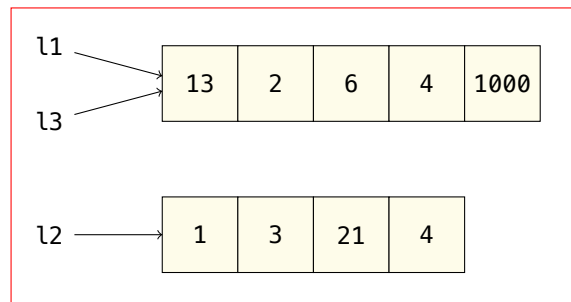
```
1 l1 = [1, 3, 6, 4]
2 l2 = l1.copy()
3 l3 = l1
4 l1[1]=2
5 l1.append(1000)
6 l2[2] = 21
7 l3[0] = 13
8 print(l1)
9 print(l2)
10 print(l3)
```

A figura a seguir mostra o estado da memória após a execução da linha 3 e ao final do programa.

Após linha 3



Após linha 6



6.2.4 Sublista

Além dos métodos mostrados, a notação usando o operador de indexação `[]` permite especificar faixa de valores para recuperar uma parte da lista, chamada **sublista**. Para recuperar uma sublista é necessário informar o índice do primeiro e do último elemento. Considere a lista a seguir.

```
l1 = [1, 3, 5, 7, 9, 2, 4, 6, 8]
```

A sublista que começa no elemento 5 (correspondente ao índice 2) e termina no elemento 4 (índice 6) é recuperada com o operador de indexação `l1[2:6]`. O python gera uma nova lista com os elementos da sublista de `l1` a partir dos índices informados. A nova lista contém uma cópia dos elementos de `l1`.

```
l1 = [1, 3, 5, 7, 9, 2, 4, 6, 8]
l2 = l1[2:6]
print(l1)
print(l2)
```

6.3 String e List

Há uma similaridade muito grande entre *string* e *List*. Uma *string* pode ser vista como uma *List* de símbolos, ou caracteres. Considere a *string* `s = "Olá mundo!"`. Esta *string* pode ser vista como uma *List* que armazena 10 valores, cada valor sendo uma letra. Espaço em branco e pontuações também ocupam espaço. A função `len(str)` retorna a quantidade de símbolos contidos na *string*.

0	1	2	3	4	5	6	7	8	9
O	l	á		m	u	n	d	o	!

Dessa forma o acesso a partes da *string* é feito de forma similar ao acesso a elementos de uma *List*, usando o operador de indexação: `[ind]`. Execute e observe o resultado do trecho de código a seguir:

```
mensagem = "Olá mundo!"
x = len(mensagem)
print(x)
print(mensagem[4])
print(mensagem[4,5])
```

6.3.1 Conversão `List` \leftrightarrow `string`

Para gerar, em python, uma `List` de caracteres usa-se a função `list()`, como mostrado no trecho de código a seguir.

```
mensagem = "Tecnologia em análise e desenvolvimento de sistemas"
letras = list(mensagem)
qtd=len(letras)
print(qtd)
print(letras[0])
print(letras[qtd-1])
```

Este programa transforma a *string* armazenada na variável `mensagem` em uma lista de caracteres. O programa então mostra a quantidade de caracteres na lista, a primeira letra e a última letra.

Para criar uma *string* a partir de uma lista de *strings*¹ usa-se o método `join` de uma string. Este método permite criar uma *string* concatenando todas as strings da lista. O método é aplicado em uma *string* usada como separador das *strings* a serem concatenadas. Por exemplo, considere a lista `l1=['a','b','c']`. É possível criar uma *string* a partir dos elementos da lista. Observe o trecho de código a seguir:

```
l1=['a','b','c']
s1 = ",".join(l1)
s2 = "".join(l1)
print(s1)
print(s2)
```

Execute este programa e veja a saída. A variável `s1` contém a *string* `"a,b,c"`, pois o método `join()` concatena os elementos de `l1` adicionando a vírgula entre cada elemento. Você consegue saber qual *string* está armazenada na variável `s2`?

6.4 Exercícios

1. Escreva um programa que leia uma lista de números e mostre o primeiro número na lista.

Exemplo de entrada e saída para a execução do programa:

¹Observe que uma lista de caracteres é uma lista de *strings* onde cada string contém exatamente um caractere.

Exemplo de entrada 1	Exemplo de saída 1
10 20 30	10

Exemplo de entrada 2	Exemplo de saída 2
50 10 43 12 23 43 12 56 78 12	50

2. Escreva um programa que leia uma lista de números e troque o primeiro elemento com o segundo. Depois mostre a nova lista.

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
10 20 30	20 10 30

Exemplo de entrada 2	Exemplo de saída 2
2 1 3 4 5 6	1 2 3 4 5 6

3. Escreva um programa que leia uma lista de 5 (cinco) números inteiros e mostre o primeiro e o último número.

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
1 2 3 4 5	1 5

Exemplo de entrada 2	Exemplo de saída 2
10 8 6 4 2	10 2

4. Escreva um programa que leia uma lista de 8 números e troque de lugar o primeiro número e o último número. Mostre a lista modificada.

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
1 2 3 4 5 6 7 8	8 2 3 4 5 6 7 1

Exemplo de entrada 2	Exemplo de saída 2
10 8 6 4 2 1 3 5	5 8 6 4 2 1 3 10

5. Escreva um programa que leia uma lista de números inteiros em uma linha, depois leia um número inteiro e adicione o número lido ao final da lista. Mostre a nova lista na tela.

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9

Exemplo de entrada 2	Exemplo de saída 2
1 3 4 2 5 8	1 3 4 2 5 8

6. Escreva um programa que leia uma lista de números, separados por espaço, e mostre a quantidade de números lidos.

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
10 20 30	3

Exemplo de entrada 2	Exemplo de saída 2
50 10 301232131 2032010 1	5

7. Escreva um programa que leia uma lista de números, separados por espaço, mostre a quantidade de números lidos, o primeiro número e o último número da lista.

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
10 20 30	3 10 30

Exemplo de entrada 2	Exemplo de saída 2
50 10 301232131 2032010 1	5 50 1

8. Escreva um programa que leia uma lista de números inteiros e mostre o menor número e o maior número da lista.

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
1 2 3 4 5 6 7 8	1 8

Exemplo de entrada 2	Exemplo de saída 2
2 3 5 4 1	5 1

9. Escreva um programa que leia uma lista de números inteiros e mostre se a soma dos números é par ou ímpar.

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
1 2 3 4 5 6 7 8	Par

Exemplo de entrada 2	Exemplo de saída 2
2 3 5 4 1	Ímpar

10. Escreva um programa que leia uma lista de números inteiros e mostre se a soma dos números é divisível pela quantidade de números lidos.

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
3 6 3	Sim

Exemplo de entrada 2	Exemplo de saída 2
1 2 3 4	Nao

11. Escreva um programa que leia uma lista de números, separados por espaço, e mostre o número localizado no meio da lista. Caso a lista tenha uma quantidade par de números o programa mostra a média dos dois números do meio.

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
10 20 30	20

Exemplo de entrada 2	Exemplo de saída 2
50 10 301232131 2032010 1	301232131

Exemplo de entrada 3	Exemplo de saída 3
1 3 5 7 9 2 4 6 8 10	5.5

12. Escreva um programa que leia uma *string* e mostre quantas palavras existem na *string*.

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
Bom dia	2

Exemplo de entrada 2	Exemplo de saída 2
TADS é muito legal	4

Exemplo de entrada 3	Exemplo de saída 3
Independencia ou morte	3

Capítulo 7

Funções

Programas de computador podem ter milhares, mesmo milhões, de comandos a serem executados. É comum que o mesmo trecho de código possa ser necessário em diferentes pontos do programa. Não é produtivo escrever, ou mesmo copiar e colar, um trecho de código cada vez que você tiver a necessidade de usá-lo. **Funções** são trechos de código, que possuem um nome, e podem ser usados em diferentes pontos do programa. Este capítulo mostra como construir funções em Python, detalhando os elementos necessários e como usar.

7.1 Introdução

O uso do mesmo trecho de código em diferentes partes de um programa pode gerar um problema de gerenciamento, pois além da quantidade de código ter uma tendência a crescer muito, a modificação, seja para correção de erros, inclusão de nova funcionalidade do trecho ou mesmo otimização, torna a tarefa de manutenção do programa praticamente inviável. Para resolver este problema as linguagens de programação permitem a definição de **funções**, onde é dado um nome a um trecho de código e o mesmo pode ser utilizado em qualquer parte de qualquer programa simplesmente especificando o nome do trecho de código.

Definição 1. *Uma **função** é um bloco/trecho de código, que possui um **nome**, e pode ser executado a partir de diferentes pontos do programa.*

Você já tem usado funções desde os primeiros programas python: `print()`, `input()` e `len()` são apenas alguns exemplos de funções. Uma função pode receber argumentos, que são valores informados na chamada à função, para que os mesmos sejam usados durante a execução do trecho de código contido na função, detalhados

na seção 7.2.2. Após execução, a função pode retornar um valor, usado pelo trecho de código que chama a função para realizar seus cálculos, o retorno será detalhado na seção 7.2.1. Um exemplo completo de função é a função que calcula a **raiz quadrada** de um número. Esta função recebe como argumento o número para que a função calcule a raiz quadrada e tem como retorno a raiz do número passado como argumento.

Considere um programa que leia as constantes de uma equação de segundo grau a , b e c e mostre a quantidade de raízes reais distintas e o valor de cada uma das raízes, mostrando a seguir.

```
1 import math
2 a,b,c=map(float,input().split())
3 delta=b*b-4*a*c
4 if (delta<0):
5     qtd_raizes=0
6     resposta="Nenhuma raiz real!"
7 elif(delta==0):
8     qtd_raizes=1
9     x1=-b/2*a
10    resposta="Uma raiz real: "+str(x1)
11 else:
12    raiz_delta = math.sqrt(delta)
13    x1=(-b+raiz_delta)/2*a
14    x2=(-b-raiz_delta)/2*a
15    resposta= "Duas raízes reais: "+str(x1)+" e "+str(x2)
16 print(resposta)
```

Observe que na linha 12 o cálculo da raiz quadrada é necessário, mas isso é feito realizando uma **chamada a função** `math.sqrt()`.

O cálculo da raiz quadrada é um trecho de código mais complexo e que pode ser usado em diferentes programas, ou mesmo em diferentes partes de um programa. Além disso, o trecho de código acima, calcular as raízes de uma equação, ficaria confuso caso o cálculo de uma raiz quadrada estivesse misturado com o resto, dificultando o entendimento e a manutenção do código.

Quanto a manutenção, fica também bastante complexo gerenciar e corrigir erros/*bugs* quando o mesmo trecho de código se repete em vários lugares. Dessa forma, o trecho em questão é colocado a parte e um nome lhe é dado, para ser usado de qualquer outra parte do programa, ou mesmo em outro programa.

7.2 Definição em Python

A definição de uma função começa com a palavra **def** seguida do nome da função, como mostrado no trecho de código a seguir, que define uma função para somar 2 números, chamada `soma()`.

```
1 def soma(a,b):  
2     c = a+b  
3     return c
```

Depois do nome da função está a definição dos parâmetros, entre parênteses. Nessa função foram definidos 2 parâmetros, *a* e *b*. Após os parâmetros coloca-se os dois-pontos (`:`) e as linhas a seguir contém o trecho de código da função, chamado de **corpo da função**. Deve ser indentado, indicando que pertence à definição da função. A última linha da função `soma()` contém uma instrução de retorno, indicando que a função retorna o valor armazenado na variável *c* para o seu chamador. O código da função só é executado quando a mesma é chamada. A definição apenas armazena o trecho de código para poder ser chamado de outras partes do programa, ou de outros programas.

O trecho de código a seguir chama a função passando os valores 10 e 20 como argumentos.

```
1 s = soma(10,20)  
2 print(s)
```

Na linha 1 há uma atribuição à variável *s*. O lado direito da atribuição contém uma expressão, onde uma chamada a uma função pode ser usada como operando da expressão. O valor retornado pela função é substituído pela chamada na expressão. No caso da atribuição à variável *s*, a expressão é composta unicamente pela chamada à função.

A seguir o código completo que lê 2 (duas) notas da entrada padrão e mostra a média das mesmas. A primeira com peso 2 e a segunda com peso 3. O cálculo da média foi colocado em uma função para organizar melhor o programa.

```
1 def calcula_media(a,b):  
2     total = a*2+b*3  
3     media=total/5  
4     return media  
5  
6 n1,n2 = map(float,input().split())  
7 m = calcula_media(n1,n2)
```

```
8 print("{:.1f}".format(m))
```

A função é **definida** entre as linhas 1 e 4. A execução do programa começa na linha 6. A função `calcular_media()` é chamada na linha 7, quando o código da função é executado. Na linha 8 o programa mostra na tela o valor da média calculada.

7.2.1 Retorno

O retorno da função é um valor colocado em substituição da chamada da função durante a execução do programa. Seja `maior(a,b)` uma função que determina o maior de 2 números. O retorno pode ser usado em qualquer lugar que aceite dados do tipo de retorno da função.

Considere o cálculo da média onde existem 2 notas para o segundo bimestre e o sistema deve usar a maior das duas. A entrada é composta de 3 notas, $n1$, $n2$ e $n3$, sendo $n1$ a nota do primeiro bimestre e a nota do segundo bimestre deve ser a maior entre $n2$ e $n3$. O trecho de código a seguir é a versão inicial do programa.

```
1 n1,n2,n3 = map(float, input().split())
2 if (n2>n3):
3     nota2=n2
4 else:
5     nota2=n3
6 media = (n1*2+nota2*3)/5
7 print("{:.1f}".format(media))
```

Observe que o trecho de código entre as linhas 2 e 5 é usado para escolher uma nota, no caso a maior entre $n2$ e $n3$. Este trecho não faz parte da semântica do cálculo da média. Para o cálculo da média é usado apenas o resultado desta escolha. Determinar o maior entre dois números é uma operação útil, que pode ser reutilizada em diferentes trechos de um programa mais complexo. Na verdade este trecho de código pode ser reutilizado em diferentes programas.

Observe o programa a seguir, onde a maior nota é determinada usando a função `maior(a,b)`.

```
1 def maior(x,y):
2     if (x>y):
3         m=x
4     else:
5         m=y
6     return m
```

```
7 |  
8 | n1,n2,n3 = map(float, input().split())  
9 | nota2 = maior(n2,n3)  
10 | media = (n1*2+nota2*3)/5  
11 | print("{:.1f}".format(media))
```

Na linha 9 a variável `nota2` recebe o valor retornado pela função. Observando a função percebe-se que o valor será sempre o maior entre os dois valores passados como argumento, que é armazenado na variável `m` na função.

Neste exemplo o valor de retorno da função é atribuído à variável `nota2`, que é usada na expressão de cálculo da média na linha 10. A função `maior` pode ser usada diretamente na expressão, já que retorna um valor real, usado no cálculo da média, como mostrado no trecho de código a seguir.

```
1 | n1,n2,n3 = map(float, input().split())  
2 | media = (n1*2+maior(n2,n3)*3)/5  
3 | print("{:.1f}".format(media))
```

Este trecho de código é mais fácil de entender e manter. A definição da função pode ser feita em outro arquivo, que não será detalhado aqui. O foco agora é entender como definir e usar funções, então não há problema em definir a função e usá-la no mesmo arquivo.

7.2.2 Parâmetros de função

Uma função recebe valores passados como **argumento** quando é chamada. A semântica de funcionamento pode variar entre linguagens de programação. É importante entender como funciona a passagem de valores para uma função. Na função o valor é recebido em uma variável chamada de **parâmetro** da função.

Definição 2. Um *parâmetro* é uma variável definida em uma função que recebe o valor a partir do trecho de código que chama a função.

Enquanto o que é passado é um **valor**, chamado de **argumento**, a variável que recebe este valor é chamada de parâmetro. No momento da chamada há uma atribuição do valor contido na código que chama ao parâmetro declarado na função.

Seja a função `f1(a,b,x)`, que retorna o valor da imagem de `x` em uma equação do primeiro grau, definida a seguir.

```
1 | def f1(a,b,x):  
2 |     r = a*x+b
```

```
3 | return r
```

e o trecho de código que chama a função

```
y=f1(10,20,2)
```

Antes de executar a primeira linha da função, há uma atribuição dos argumentos 10, 20 e 2 aos parâmetros **a**, **b** e **x**. Está implícito a execução das linhas a seguir:

```
a=10
```

```
b=20
```

```
x=2
```

Nesse caso o retorno da função é $10 \times 2 + 20$, que é **40**

Vamos agora considerar que nosso programa armazena os valores a ser passados como argumento em variáveis. É razoável pensar que os nomes das variáveis sejam **a**, **b** e **x**, assim como os nomes dos parâmetros da função. Um trecho de código pode ser visto a seguir.

```
1 | a,b,x = map(float,input().split())
2 | y=f1(a,b,x)
```

As variáveis definidas na linha 1 não fazem parte da função, portanto a variável **a** definida na linha 1 **não** é a mesma variável **a** definida como parâmetro da função, pois estão em escopos diferentes. Um escopo determina onde uma variável existe e pode ser usada. Nesse caso temos duas variáveis **a**, cada uma em um lugar diferente.

Para verificar que não são a mesma variável, você pode fazer o programa como mostrado a seguir.

```
1 | def f1(a,b,x):
2 |     r = a*x+b
3 |     return r
4 |
5 | b,a,x = map(float,input().split())
6 | y=f1(b,a,x)
7 | print(y)
```

A chamada à função, na linha 6, passa 3 argumentos, o **valor** armazenado em **b** como **primeiro** argumento, o **valor** armazenado em **a** como **segundo** argumento e o **valor** armazenado em **x** como **terceiro** argumento. Dessa forma o parâmetro **a** da função recebe o **valor** armazenado na variável **b** do programa.

Como a passagem de parâmetros é feita pela posição e o que é atribuído é o valor, uma expressão pode ser usada na chamada a função, como mostrada no trecho de código a seguir.

```
1 a,b,c,d=map(int,input().split())
2 m = maior(a*b,c*d)
```

Este trecho considera a função `maior()` definida anteriormente. A variável `m` armazena o maior entre os dois valores passados como parâmetro, nesse caso `a*b` e `c*d`.

Listas como argumentos

Uma observação deve ser feita quando passamos listas como parâmetros. Tudo que foi visto até agora considera os tipos básicos `int`, `float` e `bool`.

A passagem de listas para funções é feita por **referência**, dessa forma se uma função alterar um conteúdo de uma lista, ela altera a lista original, passada como parâmetro. Observe o trecho de código a seguir.

```
1 def f2(l,a):
2     l[0] = a
3
4 lista01 = [1,2,4,6,8,10]
5 f2(lista01,50)
6 print(lista01)
```

Nesse trecho de código há apenas uma lista, pois o parâmetro `l` da função `f2()` possui uma referência para a lista `lista01`. Ao se modificar o valor armazenado no índice 0 (zero) na função, há uma modificação na lista do trecho de código chamado.

Para se criar uma cópia é necessário realizar uma clonagem da lista, que pode ser feita usando o método `copy()` da lista:

1. Na chamada da função.

```
f2(lista01.copy())
```

2. No início da função.

```
def f2(l,a):
    nova_lista=l.copy()
    nova_lista[0] = a
```

Em ambos os casos é importante notar que a lista usada durante a execução da função deixa de existir quando a função termina. Para resolver esse problema, caso seja necessário, a função pode retornar a lista interna usada durante a execução, como mostrado no trecho a seguir.

```
1 def f2(l,a):
2     nl = l.copy()
3     nl[0] = a
4     return nl
5
6 lista01 = [1,2,4,6,8,10]
7 lista02 = f2(lista01,50)
8 print(lista01)
9 print(lista02)
```

Nessa nova versão a função `f2()` retorna uma nova lista, que é atribuída à variável `lista02`. Depois da chamada à função `f2()` o programa possui duas listas na memória, uma referenciada pela variável `lista01` e outra referenciada pela variável `lista02`.

7.3 Funções podem chamar outras funções

O uso de funções é fundamental em programas mais complexos. É através da definição e uso que o desenvolvedor **modulariza** um programa, dividindo operações complexas em um conjunto de operações mais simples.

Além da modularização, o uso de funções permite trabalhar trechos de código com objetivos específicos de forma independente. Ao se definir uma nova função pode-se **reusar** uma função já definida. Dessa forma um programa é um conjunto de funções com objetivos bem definidos que, em conjunto, realizam tarefas complexas.

Antes de continuar com o estudo de funções, vale a pena discutir **manutenção** de sistemas.

Manutenção

A manutenção consiste em alterar um programa já existente. Qualquer modificação é considerada manutenção, mas pode-se definir 3 formas distintas de manutenção:

Adaptativas Tem o objetivo de adaptar o sistema a uma nova realizada ou a novas regras externas. Por exemplo, em um sistema de contabilidade pode-se fazer manutenção adaptativa para adequar o sistema as novas regras de imposto após uma mudança na lei.

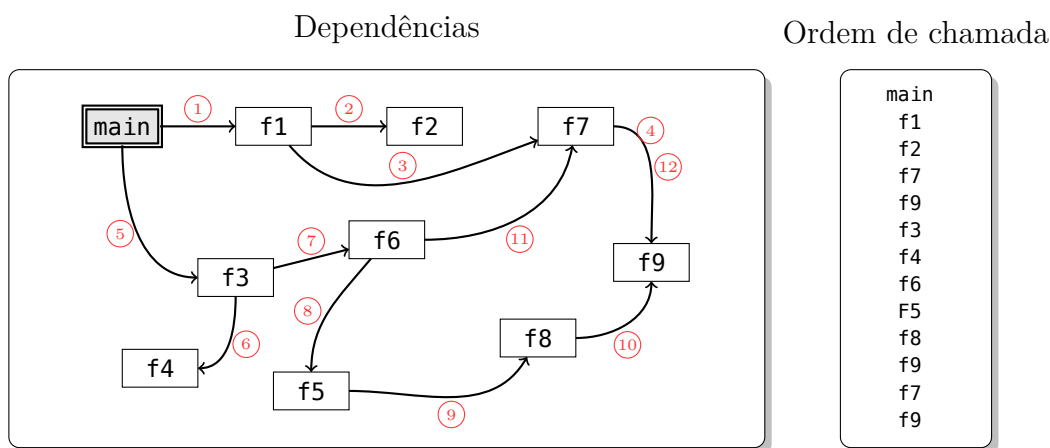
Corretiva Um erro é identificado durante os testes ou depois que o sistema é colocado em produção e precisa ser corrigido. Um erro é chamado de **bug** em

desenvolvimento de sistemas.

Evolutiva O sistema passa a fazer operações que não fazia. Inclusão de nova funcionalidade pode acontecer como melhoria do sistema, identificada através de uma nova análise de requisitos, ou de novas solicitações por parte dos usuários ou clientes.

De volta às funções, além do reuso de trechos de códigos, a divisão do programa em funções permite que a manutenção seja feita de forma mais eficiente.

Quanto à divisão do programa em várias funções, a figura a seguir mostra um conjunto de funções e suas dependências, são as funções de $f_1()$ a $f_9()$ e a função principal $main()$.



A esquerda da figura, no quadro **dependências** é mostrado quais funções dependem de quais funções, a direita, no quadro **ordem de chamada**, é mostrado a ordem de chamada das funções. A ordem em que as funções são chamadas também está ilustrado nos números ao lado das setas de dependências.

O programa começa por uma função chamada `main()`, que em Python é implícita, é o código no arquivo que você executa, código este que está definido fora de uma função.

A função `main()`, no exemplo da figura depende das funções `f1()` e `f3()`, nessa sequência. Contudo, durante a execução de `f1()`, há mais chamadas de função. A função `f2()` é chamada a partir da função `f1()`, fazendo com que `f2()` seja executada antes da função `f3()`.

Seguindo os números nas setas você pode acompanhar, além da dependência, a ordem em que as funções são chamadas. Observe que quando uma função termina sua execução, o fluxo de execução do programa volta para o ponto em que a função

foi chamada. A função `f7()` é chamada tanto pela função `f1()` quando pela função `f6()` e não tem conhecimento de onde foi chamada. O fluxo de execução é controlado pelo interpretador Python, garantindo o retorno correto para o ponto que chamou a função.

7.4 Funções recursivas

Recursividade é uma ferramenta poderosa em programação para resolver problemas. A recursividade se caracteriza quando é necessário realizar a mesma operação diversas vezes, em subproblema(s) do problema principal. Um vez resolvido o subproblema, a solução é então combinada com a instância do problema principal.

7.4.1 Comportamento recursivo

Em programação, uma função recursiva é caracterizada quando ela faz uma chamada a ela mesma. Dessa forma, é possível aplicar a mesma operação a uma instância diferente do mesmo problema. Esta instância é, normalmente, uma parte menor¹ da instância atual.

É importante salientar que a subdivisão do problema deve gerar um **caso de base** do problema, onde a solução é direta e não é necessário dividir o problema.

7.4.2 Cálculo do Fatorial

Para entender na prática o uso de recursividade vamos analisar o cálculo do fatorial de um número inteiro não negativo n , denotado por $n!$ (n seguido de um ponto de exclamação).

Para se calcular o fatorial de um número n aplica-se a definição: $n! = n \times (n-1)!$, ou seja, para calcular o fatorial de n é necessário calcular o fatorial de $n-1$, onde é necessário calcula o fatorial de $n-2$, pois $(n-1)! = (n-1) \times (n-2)!$, e assim sucessivamente. Percebe-se também que, em algum momento, esta dependência recursiva deve parar. No caso da função fatorial, o **caso de base** é o fatorial de 0, definido como 1. Pode-se dizer que:

$$n! = \begin{cases} 1 & \text{Se } n = 0 \\ n \times (n-1)! & \text{Caso contrário} \end{cases}$$

¹A noção de menor aqui é abstrata. De fato, como sera visto adiante, a mudança de instância do problema deve levar ao(s) caso(s) de base da definição recursiva.

Para se calcular o fatorial de 4 temos, inicialmente, que $4! = 4 \times 3!$. Nesse caso há uma redução do problema, no sentido que o único fatorial definido é o fatorial de 0 e, para todos os números inteiros não negativos, é necessário realizar a mesma operação para o seu antecessor, sendo o antecessor outra instância do mesmo problema. O cálculo do fatorial de 4, passo-a-passo, é mostrado a seguir:

$$\begin{aligned}
 4! &= 4 \times 3! \\
 4! &= 4 \times (3 \times 2!) \\
 4! &= 4 \times (3 \times (2 \times 1!)) \\
 4! &= 4 \times (3 \times (2 \times (1 \times \boxed{0!}))) \leftarrow \text{caso de base} \\
 4! &= 4 \times (3 \times (2 \times (1 \times 1))) \\
 4! &= 4 \times (3 \times (2 \times 1)) \\
 4! &= 4 \times (3 \times 2) \\
 4! &= 4 \times 6 \\
 4! &= 24
 \end{aligned}$$

Ao se encontrar o caso de base, o valor obtido é usado pela instância que **chamou** o calculo do fatorial com o subproblema.

Em termos de programação, uma função para calcular o fatorial pode ser definida como **fat(n)**. É possível, usando a mesma representação anterior, definir os casos da função fatorial:

$$fat(n) = \begin{cases} 1 & \text{Se } n = 0 \\ n \times fat(n - 1) & \text{Caso contrário} \end{cases}$$

O caso de base determina quando a função deve “parar” e o caso geral é responsável por extrair o cálculo da instância atual e chamar recursivamente a função com o subproblema. Toda função recursiva deve ter pelo menos um caso de base. O que é o caso de base depende da operação sendo realizada. A definição do(s) caso(s) geral(is) deve(m) obrigatoriamente levar as chamadas recursivas a um caso de base. Caso isso não ocorra a função nunca termina.

A função **fat(n)** em python é definida como:

```

1 def fat(n):
2     if (n==0):
3         return 1
4     else:
5         return n * fat(n-1)

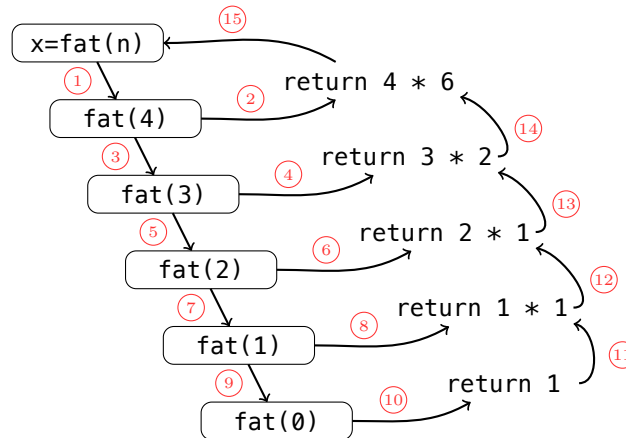
```

Cada caso analisado de uma função recursiva deve ser analisado no código através de uma condição. No exemplo do fatorial um **if** para verificar o valor de n é suficiente.

Para efeito de simplificação de código não é verificado neste exemplo se o parâmetro passado é um inteiro não negativo. Considerando a função `fat(n)` definida e o trecho de código a seguir:

```
n = int(input())
x = fat(n)
print("Fatorial de {:d} é {:d}".format(n,x))
```

Considere agora uma execução em que o usuário digita 4, o programa calcula o fatorial de 4 chamando a função `fat(n)` recursivamente, como ilustrado a seguir:



Os números nos círculos mostram a sequência de execução da chamada recursiva. A linha que chama a função recursiva, `x=fat(n)`, é o início da execução da função fatorial. O valor de n é 4, executando a função fatorial com o parâmetro 4. A função fatorial executa a linha 5, `return n * fat(n-1)`. Quando há uma chamada de função, todos os dados da função em execução ficam em uma área de memória e uma nova chamada de função cria outra área de memória, mesmo que seja a mesma função, dessa forma o retorno de uma função sempre é garantido para o ponto onde foi chamada. A função `fat(4)` é colocada em espera na linha 5, e a função `fat(3)` começa a ser executada. O mesmo acontece para todos os valores até o zero, quando a condição $n = 0$ é verdadeira, identificando o caso de base. A função `fat(0)` retorna sem chamar qualquer outra função. O retorno é feito para o ponto da função que chamou, nesse caso a função `fat(1)`, que está em espera do resultado para substituir na expressão que contém a chamada recursiva.

7.5 Exemplos de recursividade

Nesta seção será discutido algumas funções recursivas e suas implementações.

7.5.1 Exponenciação

A exponenciação, ou potenciação, em matemática consiste em multiplicar um número a por ele mesmo um determinado número b de vezes, denotado por a^b . Dessa forma 3^4 pode ser calculado multiplicando-se 3 quatro vezes: $3 \times 3 \times 3 \times 3$.

Ao se observar com mais atenção pode-se perceber que a expressão 3^4 também pode ser escrita como 3×3^3 , já que 3^3 pode ser escrita como $3 \times 3 \times 3$. A se generalizar temos que a^b pode ser escrita como $a \times a^{b-1}$, gerando uma definição recursiva da exponenciação, onde $a^b = a \times a^{b-1}$. E o caso de base? Bom, nesse caso pode-se considerar que todo número elevado a 0 (zero) é 1 (um): $a^0 = 1$. Deve-se fazer a ressalva, portanto, que esta definição considera números inteiros não negativos, já que a definição recursiva considera o subproblema com $b - 1$ e, caso números negativos sejam considerados, a função nunca se direciona ao caso de base, onde $b = 0$. Assim sendo, a função exponencial pode ser definida de forma recursiva como:

$$a^b = \begin{cases} 1 & \text{Se } b = 0 \\ a \times a^{b-1} & \text{Caso contrário} \end{cases}$$

e em notação de programação:

$$\text{exp}(a, b) = \begin{cases} 1 & \text{Se } b = 0 \\ a \times \text{exp}(a, b - 1) & \text{Caso contrário} \end{cases}$$

Com o caso de base e o caso geral definidos fica trivial a implementação em Python:

```
def exp(a,b):
    if (b==0):
        return 1
    else:
        return a * exp(a,b-1)
```

7.5.2 Divisores de um número

Dado um número inteiro n não negativo², quais são os divisores de n ? Antes de identificar todos os divisores de n começamos por saber como identificar se um número d é divisor de n . Essa operação consiste em realizar uma divisão inteira entre n e d e verificar o valor do resto. Caso o resto seja 0 (zero) então d é divisor de n . Dessa

²Será usado não negativos para simplificação do código.

forma, para enumerar os divisores de um número n é necessário verificar se o resto da divisão entre n e d é zero para cada divisor em potencial.

Os divisores potenciais de n , em uma primeira análise, e considerando apenas números não negativos, são todos os números entre 1 e o próprio n . Por exemplo, para o número 20 os possíveis divisores são:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19 e 20,

pois nenhum número acima de 20 pode ser divisor de 20. Os divisores de 20 são: 1, 2, 4, 5, 10 e 20, sendo um total de 6 divisores.

Quantidade de divisores

Primeiramente vamos definir uma função para calcular a quantidade de divisores de um número n . Para definir uma função recursiva é necessário identificar o caso geral e o caso de base. Temos que o valor do número n a ser verificado não muda. A mudança de instância na chamada recursiva se dá pelo potencial divisor d . Então vamos considerar que para um valor d a função retorna a quantidade de divisores de n entre 1 e d . Por exemplo, para $n = 20$ e $d = 4$ a função deve retornar 3, já que existem 3 divisores de 20 entre 1 e 4, incluindo o 4. A definição da função pode ser feita verificando se d é divisor de n e adicionar o resultado à chamada recursiva para os valores menores que d , ou seja, para a instância $d - 1$. Fica fácil identificar o caso de base como $d = 1$, pois há exatamente 1 divisor de n no conjunto unitário contendo o número 1.

Em termos matemáticos podemos definir a função recursiva como:

$$\text{qtd_div}(n,d) = \begin{cases} 1 & \text{Se } d = 1 \\ \text{qtd_div}(n,d) & \text{Se } n \% d \neq 0 \\ 1 + \text{qtd_div}(n,d) & \text{Se } n \% d = 0 \end{cases}$$

A seguir a definição da função em Python:

```
def quantidade_divisores_rec(n,d):
    if (d==1):
        return 1
    elif (n%d!=0):
        return quantidade_divisores_rec(n,d-1)
    else:
        return 1+quantidade_divisores_rec(n,d-1)
```

Esta implementação é uma tradução direta do modelo matemático para Python. Podemos melhorar um pouco a qualidade dessa implementação. Primeiro separando

caso de base e geral em um `if..else.`, depois realizando apenas uma chamada recursiva, pois é possível observar que as chamadas recursivas na função anterior são idênticas, além de que apenas uma delas é de fato executada. Depois desta análise podemos implementar a função a seguir:

```
def quantidade_divisores_rec(n,d):  
    if (d==1):  
        return 1  
    else:  
        divisor = 0  
        if (n%d==0):  
            divisor = 1  
        return divisor + quantidade_divisores_rec(n,d-1)
```

Esta implementação é semanticamente igual a anterior, mas com uma melhor organização de código com relação às condições e a chamada recursiva. Além da organização do código em si há ainda um pequeno problema da assinatura da função, com dois parâmetros. Observe que quando se pergunta quantos divisores tem um número, há apenas um parâmetro, que é o próprio número n . O segundo parâmetro foi adicionando para permitir uma mudança de instância na chamada recursiva. Quando há essa necessidade, é comum a implementação de 2 funções, uma com a interface do problema, nesse caso contendo apenas um parâmetro, e outra com a interface da definição recursiva, com 2 parâmetros. A função com a interface padrão é definida a seguir:

```
def quantidade_divisores(n):  
    return quantidade_divisores_rec(n,n)
```

A implementação de 2 (duas) funções facilita o uso, pois o programador usa a função pública, que possui apenas o(s) parâmetro(s) da interface pública da função.

Lista de divisores

Usando a mesma lógica da quantidade de divisores é possível criar também uma função que retorne uma lista com todos os divisores de n . O código da função, em Python, é mostrado a seguir:

```
def divisores_rec(n,d):  
    if (d==1):  
        return [1]  
    else:  
        divs = divisores_rec(n,d-1)
```

```

if (n%d==0):
    divs.append(d)
return divs

```

No caso da linguagem Python, o retorno de uma lista é de implementação trivial. Ao se considerar outras linguagens, como **C/C++** pode haver necessidade de implementar o retorno de forma otimizada, usando ponteiros ou classes de bibliotecas, como o `std::vector` em **C++**.

Assim como na contagem, para facilitar o uso da função é adequado implementar uma função para uso pelo programador, sem o parâmetro de controle da recursão:

```

def divisores(n):
    return divisores_rec(n,n)

```

A função retorna uma lista com todos os divisores de n . A seguir o programa completo que lê um número n digitado pelo usuário e mostra todos os divisores de n .

```

def divisores_rec(n,d):
    if (d==1):
        return [1]
    else:
        divs = divisores_rec(n,d-1)
        if (n%d==0):
            divs.append(d)
        return divs

def divisores(n):
    return divisores_rec(n,n)

num = int(input())
divisores = divisores(num)
print("O número {:d} possui {:d} Divisores: ".format(num,len(divisores)),end="")
print(divisores)

```

7.5.3 Sequencia de Fibonacci

A sequencia de Fibonacci (https://pt.wikipedia.org/wiki/Sequ%C3%A2ncia_de_Fibonacci) é uma sequencia infinita de números onde cada elemento da sequencia é obtido a partir dos seus dois antecessores o n -ésimo elemento é definido por

$F_n = F_{n-1} + F_{n-2}$. A sequência possui dois casos de base, que são o primeiro e segundo elementos. O primeiro elemento é o número 0 (zero) e o segundo é o número 1 (um)³. Dessa forma podemos definir a sequência como:

$$F_n = \begin{cases} 0 & \text{Se } n = 1 \\ 1 & \text{Se } n = 2 \\ F_{n-1} + F_{n-2} & \text{Se } n > 2 \end{cases}$$

Segue a conversão para a linguagem Python:

```
def fibonacci(n):
    if (n==1):
        return 0
    elif (n==2):
        return 1
    else:
        return fibonacci(n-1)+fibonacci(n-2)
```

É importante observar que a função recursiva realiza 2 (duas) chamadas a cada vez, cada chamada com um parâmetro diferente, $n - 1$ e $n - 2$. Esta propriedade é própria da função fibonacci e leva uma quantidade de chamadas a função muito alta. Por esta razão o cálculo de números de fibonacci com valores maior do que 30 começa a ficar lento. Há como melhorar o desempenho desta função, mas foge ao escopo deste material. As duas técnicas usadas são *memoization* e/ou Programação dinâmica.

7.6 Recursividades em listas

Uma lista é uma coleção de elementos onde a localização de cada elemento é um índice, que é número inteiro. Ao se trabalhar com coleções é comum a necessidade de realizar alguma operação com todos os elementos da coleção, um por um. Nesta sessão discutiremos como usar a recursividade para trabalhar com cada elemento da lista, um por vez, além de usar o resultado de forma generalizada para todos os elementos da lista.

Para usar a recursividade e trabalhar com todos os elementos de uma lista devemos primeiro definir qual elemento trabalhar da lista. Vamos considerar sempre o

³Há também versões que consideram o primeiro e o segundo elementos da sequência como sendo 1. Dessa forma $F_1 = 1$ e $F_2 = 1$. Ou começando os elementos da sequência em 0 (zero), sendo $F_0 = 0$ e $F_1 = 1$, e a recursão começa no termo $F_2 = F_0 + F_1$.

último elemento da lista como sendo nosso elemento de trabalho. Quando se considera o último elemento como o de trabalho podemos observar que temos o elemento atual e uma lista com todos os elementos menos o atual. Por exemplo, considere a lista $[3, 5, 2, 4, 9, 6]$. Podemos definir o elemento 6 como de trabalho, que está no índice 5. Também podemos considerar que há uma lista que contém todos os elementos menos o elemento 6: $[3, 5, 2, 4, 9]$. Ao se usar a recursividade é possível sempre trabalhar com o último elemento da lista e chamar recursivamente passando a lista como parâmetro, dessa vez sem o último elemento. O caso de base será a lista vazia ou a lista com um elemento, dependendo do problema.

7.6.1 Soma dos elementos de uma lista

No caso de uma lista, usando recursividade, podemos considerar que a soma de todos os elementos de uma lista é a soma do último elemento com a soma da sub-lista contendo todos os elementos menos o último. O caso de base pode ser considerado quando a lista não possui elementos. Nesse caso a soma é 0 (zero). Considere a lista X de n elementos representada por $\{x_i\}_{i=0}^{n-1}$. O primeiro elemento é o x_0 , o segundo $x_1 \dots$ e o último x_{n-1} . A quantidade de elementos de X é definida como $|X|$. Podemos definir a soma de todos os elementos como:

$$\sum_0^{n-1} x_i = \begin{cases} 0 & \text{Se } |X| = 0 \\ x_{n-1} + \sum_0^{n-2} x_i & \text{Se } |X| \geq 1 \end{cases}$$

Ou seja, a soma dos elementos de uma lista X é o último elemento somado a soma dos elementos da sub-lista de X que contém todos os elementos menos o último.

Para implementar em python esta função é necessário antes realizar a operação de recuperar a sub-lista sem o último elemento. Começaremos pela forma mais simplificada e depois evoluímos. Para remover o último elemento de uma lista pode-se usar o método `pop()`, que remove o último elemento da lista. Para que não se perca a lista original devemos realizar uma cópia da lista e então remover o último elemento. Segue a primeira versão da nossa função recursiva que soma os elementos de uma lista x em python:

```
def soma_rec(x):
    tamanho = len(x)
    if (tamanho==0):
        return 0
    else:
        sublista = x.copy()
```

```

sublista.pop()
return x[tamanho-1]+soma_rec(sublista)

```

Podemos simplificar mais ainda nossa função se usarmos operações de divisão de listas em python (*Python List Slicing*). Para realizar uma cópia da lista l e remover o último elemento ao mesmo tempo podemos escrever $l[:-1]$. Segue o código simplificado:

```

def soma_rec(x):
    tamanho = len(x)
    if (tamanho==0):
        return 0
    else:
        return x[tamanho-1]+soma_rec(x[:-1])

```

Observe que o código é uma tradução direta do modelo matemático definido anteriormente.

NOTA: É importante ressaltar que usar a função `sum(l)` é mais eficiente do que a função que acabamos de definir. O objetivo aqui é a compreensão de funções recursivas. Também é importante ressaltar que há formas de se otimizar funções quando o uso da recursividade demanda muito da memória e da pilha de chamadas do processo. Para minimizar o uso da pilha de chamadas, reduzindo a memória necessária para a execução, usa-se laço ao invés de chamadas recursivas. Laços serão visto no capítulo ??.

7.6.2 Maior elemento de uma lista de inteiros

Considere agora o problema de identificar qual o maior elemento de uma lista de números inteiros. Se a lista tiver apenas 1 (um) elemento, o maior é o próprio elemento, caso contrário usamos a recursividade para identificar qual o maior elemento entre o último elemento e o maior elemento da sub-lista que contém todos os elementos menos o último. Podemos partir direto para a implementação em Python:

```

def maior_rec(l):
    if len(l)==1:
        return l[0]
    maior_elemento = maior_rec(l[:-1]) # <-- Chamada recursiva
    if (l[-1]>maior_elemento):          # l[-1] pega o último elemento da lista l
        maior_elemento = l[-1]         # equivale a l[len(l)-1]
    return maior_elemento

```

Para facilitar a leitura e entendimento do código podemos criar uma função que retorne o maior entre dois elementos e modificar a função `maior_rec()` para usar essa função, como mostrado a seguir:

```
def maior2(a,b):
    if (a>b):
        return a
    else
        return b

def maior_rec(l):
    # l DEVE ser uma lista
    if len(l)==1:
        return l[0]
    else:
        return maior2(l[-1],maior_rec(l[:-1])) # retorna maior entre último
                                                # e maior da sub-lista l[:-1]
```

7.6.3 Como evitar cópias de listas

Um problema que deve ser considerado quando usamos recursividade e copiamos listas é que o uso de memória aumenta consideravelmente. Observe que cada chamada mantém os dados locais em memória e, para a cópia de lista, cada chamada da função mantém uma cópia da lista enquanto aguarda o retorno da chamada recursiva.

Vamos imaginar uma chamada para identificar o maior elemento da lista passando como argumento uma lista de 1000 (mil) elementos. Para calcular o maior, é feita uma cópia da lista sem o último elemento, que contém 999 elementos. Enquanto a cópia é passada como argumento, o parâmetro que recebeu a lista com 1000 elementos ocupa a memória e não é usada nas chamadas recursivas subsequentes. Até chegar no caso de base, o programa terá em memória 1000 listas, a primeira com 1000 elementos, a segunda com 999, a terceira com 998 e assim sucessivamente até a milésima lista com 1 elemento. A memória total necessária para identificar o maior elemento de uma lista de 1000 elementos é de 500500 elementos, calculado como soma dos termos da P.A. $\{ 1, 2, 3, 4, 5, \dots, 999, 1000 \}$. Além dessa memória ainda tem a memória necessária para outras variáveis da função e dados da própria função, como posição de retorno e valor de retorno.

Quando se quer evitar a cópia de lista é comum usar a mesma lista para todas as chamadas recursivas, nesse caso cria-se mais um parâmetro da função, que é a indicação de qual elemento deve ser considerado na instância da chamada recursiva. Assim, na chamada principal esse valor é o índice do último elemento e diminui em 1

para cada chamada recursiva. Também para facilitar uso define-se mais uma função sem esse parâmetro como assinatura pública da função, como mostrado a seguir:

```
def maior_rec(l,ult):                                # l DEVE ser uma lista
    if ult==0:
        return l[0]
    else:
        return maior2(l[ult],maior_rec(l,ult-1))    # retorna maior entre último
                                                    # e maior da sub-lista l[:-1]

def maior_lista(l):
    return maior_rec(l,len(l)-1)
```

7.7 Exercícios

1. Seja o programa a seguir.

```
def f1(a,b):  
    return a*2+b*3  
  
x,y = map(int,input().split())  
print(f1(10,20))  
print(f1(x,10))  
print(f1(20,y))  
print(f1(x,y))
```

Escreva a saída do programa para os valores de entrada:

- 100 e 200
- 10 e 20
- 2 e 3

2. Considere o programa a seguir.

```
def f1(a,b):  
    if a>b:  
        return b  
    else:  
        return a  
  
def f2(a,b,c):  
    x = f1(b,c)  
    y = x+f1(c,a)  
    return x+y  
  
x,y,z = map(int,input().split())  
print(f2(x,y,z))
```

Escreva a saída do programa para os valores de entrada:

- 10, 20 e 15
- 20, 10 e 30

- 3, 1 e 2

3. Seja a função recursiva definida a seguir.

```
def f_rec(x):  
    if x==0:  
        return 0  
    s = x%10  
    n = x//10  
    return s+f_rec(n)
```

- Identifique o que faz a função.
- Determine se há valores em que a função nunca termina.
- Escreva o valor de retorno para cada um dos valores de entrada a seguir:
 - 8
 - 19
 - 123
 - 8192
 - 1000001

4. Seja a função recursiva definida a seguir.

```
def f(x,y):  
    if (y==0):  
        return 0  
    else:  
        return x+f(x,y-1)
```

- Determine o valor de retorno para as seguintes chamadas da função:
 - $f(2,5)$
 - $f(3,3)$
 - $f(5,3)$
 - $f(12,13)$
 - $f(11,15)$

5. Seja a função recursiva definida a seguir.

```
def f(a,b):
    if (a>=b):
        return (a+b)//2
    else:
        return f(f(a+2,b-1),f(a+1,b-2))
```

Determine o valor de retorno para a chamada $f(1,5)$. Para facilitar, faça um desenho de todas as chamadas com seus respectivos valores.

6. Você foi contratado por uma empresa de construção para fazer um programa que determine a maior área entre 4 terrenos que a empresa está sondando para construir um novo conjunto de casas. Para cada terreno você tem a largura L e a profundidade P do terreno. Seu programa deve mostrar qual terreno possui a maior área. A entrada é composta de 4 linhas, correspondente a cada um dos terrenos. Em cada linha há dois números reais L e P . Seu programa deve mostrar em uma única linha o terreno com maior área. Os terrenos são identificados pelas letras A, B, C e D. Se dois terrenos tiverem a mesma área, o programa deve mostrar o primeiro que aparece na lista.

Escreva funções para facilitar o desenvolvimento do programa. Faça o cálculo da área do terreno em uma função.

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
10 20 20 30 30 25 20 15	C

Exemplo de entrada 2	Exemplo de saída 2
14 18 14 12 12 15 12 20	A

Exemplo de entrada 3	Exemplo de saída 3
10 24 12 22 14 20 16 18	D

7. Escreva um programa que mostre se um aluno do IFRN foi aprovado ou está em recuperação. A nota do primeiro bimestre é composta de uma lista de exercício e 3 avaliações. A lista de exercício compõe 50% da nota do bimestre. Entre as 3 avaliações elimina-se a menor nota e se faz uma média aritmética simples, que corresponde aos outros 50% do bimestre. Para o segundo bimestre a lista de exercício compõe 20% da nota e há 5 avaliações, que compõe os outros 80% da nota do bimestre. A nota da 5 avaliações é calculada eliminando-se as notas extremas, a maior e a menor, e realizando-se uma média aritmética entre as 3 restantes.

A primeira linha da entrada contém as notas do primeiro bimestre, L , A_1 , A_2 e A_3 , sendo L a lista de exercícios seguida das notas das 3 avaliações. A segunda linha contém as notas do segundo bimestre L , A_1 , A_2 , A_3 , A_4 e A_5 . Todas as notas são valores inteiros entre 0 e 100 ($0 \leq L, A_1, A_2, A_3, A_4, A_5 \leq 100$). O programa deve mostrar na saída padrão uma única linha com a mensagem APROVADO OU EM RECUPERACAO.

A média M para aprovação é

$$M = \frac{B_1 \times 2 + B_2 \times 3}{5}$$

onde: B_1 é a nota do primeiro bimestre e B_2 a nota do segundo bimestre.

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
90 20 60 80 50 20 40 60 80 100	APROVADO

Exemplo de entrada 2	Exemplo de saída 2
20 20 30 70 40 20 30 40 50 60	NAO APROVADO

8. Escreva um programa que leia dois números inteiros não negativos e mostre a soma dos fatoriais do mesmo ⁴.

Escreva uma função recursiva para calcular o fatorial de um número.

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
4 4	48

Exemplo de entrada 2	Exemplo de saída 2
0 0	2

Exemplo de entrada 3	Exemplo de saída 3
0 2	3

9. A função de potenciação a^e pode ser escrita como $a \times a^{e-1}$ para todo valor de e maior-igual a 1. Para e igual a zero a função retorna 1. Esta propriedade pode ser descrita como na equação a seguir.

$$a^e = \begin{cases} 1 & \text{se } e = 0 \\ a \times a^{e-1} & \text{senão} \end{cases}$$

Escreva um programa que leia 2 (dois) números inteiros não negativos a e e da entrada padrão e escreva o valor de a^e na saída padrão.

Exemplo de entrada e saída para a execução do programa:

⁴Baseado no problema URI 1161

Exemplo de entrada 1	Exemplo de saída 1
2 10	1024

Exemplo de entrada 2	Exemplo de saída 2
5 4	625

10. Escreva um programa que leia um número inteiro não negativo e escreva a quantidade de vezes que cada dígito ocorre no número. Escreva um função recursiva para contar a quantidade de ocorrência de um dígito d em um número n . A assinatura da função é:

```
def conta_digitos(n,d):
```

Exemplo de entrada e saída para a execução do programa:

Exemplo de entrada 1	Exemplo de saída 1
12341	0: 0 1: 2 2: 1 3: 1 4: 1 5: 0 6: 0 7: 0 8: 0 9: 0

Exemplo de entrada 2	Exemplo de saída 2
8192	0: 0 1: 1 2: 1 3: 0 4: 0 5: 0 6: 0 7: 0 8: 1 9: 1

Exemplo de entrada 3	Exemplo de saída 3
81921024901	0: 2 1: 3 2: 2 3: 0 4: 1 5: 0 6: 0 7: 0 8: 1 9: 2