CI-240: Fundamentos de Programação
Notas de Aula

Prof. Armando Luiz N. Delgado

Fevereiro 2017

Conteúdo

Tópico 1

O Computador, Algoritmos e Programas

Aula 1

1.1 Apresentação do Curso

- Acessar com regularidade a página da disciplina e a página da respectiva turma .
- Verificar COM ATENÇÃO as datas de avaliações e trabalhos de sua turma (**finais e 2ª chamada inclusive**). ESTAS DATAS NÃO SERÃO ALTERADAS, a menos que ocorra um evento inesperado.
- Verificar bibliografia e material de apoio .
- O aluno deve observar com cuidado o limite de faltas, pois será feito controle rigoroso de presença (vide critérios de avaliação).
- Verificar o plano geral do conteúdo da disciplina . Recomenda-se ENFATICAMENTE que o aluno pratique a programação através do uso de notebooks/netbooks fora e, PRINCIPALMENTE, dentro da sala de aula. Esta prática é essencial à compreensão dos conceitos desenvolvidos na disciplina.

1.2 Introdução ao computador

Modelo simplificado de um processador digital:

- Estrutura de um processador digital: processador+pc+memória+periféricos
- Modelo Von-Neumann: a idéia "dados+programa em memória"
- Instruções de um processador (código de máquina) e o ciclo de execução de instruções
- Com excessão do código de máquina (que é o programa, propriamente dito), **o resto é notação**.
- Conceitos fundamentais em programação:

  - Sistemas de numeração
  - algoritmo
  - programa
  - linguagem de programação
  - compilador
  - programação modular simples

- Introdução ao conceito de algoritmo: descrição passo a passo da solução de um problema, e especificação de recursos (variáveis e tipos) necessários para resolver o problema.

Tópico 2

Primeiros Elementos da Linguagem Python

Aula 2

2.1 Introdução

O objetivo desta aula é apresentar programas simples em Python para familiarizar os alunos com a linguagem. São introduzidos também os conceitos de variáveis, atribuição e expressões aritméticas.

2.2 Programas em Python

Essencialmente, um programa Python consiste de uma ou mais linhas chamadas sentenças. Além disso, um programa em Python pode definir pelo menos uma função em qualquer local do texto do programa. O ponto de início de execução do programa é a primeira sentença do programa que está fora do corpo de definição das funções.

Programas Python tem a seguinte estrutura mínima inicial:

```
sentença 1
sentença 2
⋮
```

Figura 2.1: esqueleto1

Se o programa vai ler valores do usuário ou exibir mensagens e resultados, além disso é preciso:

```
# coding=latin-1

sentença 1
sentença 2
⋮
```

Figura 2.2: esqueleto2

Se o programa vai usar alguma função da biblioteca matemática (que será com freqüência o caso nesse curso) e da biblioteca de manipulação de texto (*strings*), além disso é preciso:

```
# coding= latin-1

from math import *
from string import *

sentença 1
sentença 2
⋮
```

Figura 2.3: esqueleto3

Assim, fica *padronizado* que a estrutura mínima de um programa em Python para os fins desta disciplina é o definido em `esqueleto3.py` (Figura 2.3)

De uma maneira geral, a estrutura *completa* de um programa em Python é:

```
# coding=latin-1

from math import *
from string import *

definição de funções
⋮

# Início do programa principal

entrada de valores para variáveis (definição de variáveis)
⋮
sentenças
⋮
exibição de resultados
⋮
```

Figura 2.4: esqueleto4

Cada um dos elementos desta estrutura geral será explicada a partir de agora.

2.3 Sentenças

Uma *sentença* (do inglês *statement*) representa um processamento que deve ser feito pelo programa. Este processamento pode ser um cálculo cujo resultado é armazenado em meória ou exibido na tela do computador ou a execução de outro programa.

Exemplos de sentenças incluem:

```
x = 3
i = float(input())
print (i,x)

i = i + 1
```

2.4 Mostrando mensagens na tela

Em Python a maneira mais simples de exibir alguma coisa na tela é utilizando a primitiva `print`. Veja o exemplo `hello.py` (Figura 2.5), que imprime Alô todo mundo! em uma linha na tela do computador. Após a impressão da linha, o cursor de texto se posiciciona no início da linha seguinte.

```
#! /usr/bin/env python3
# coding=latin-1

from string import *
from math import *

print ('Alô todo mundo!')
```

Figura 2.5: hello.py

O próximo passo é fazer um programa que exibe na tela um valor ou conjunto de valores que foram calculados durante o programa.

Para isto, devemos introduzir e entender com mais detalhes o conceito de **variável** em uma linguagem de programação, visto superficialmente na aula 1 .

2.5 Variáveis e Tipos

Uma variável é uma informação que você pode usar dentro de um programa Python . Esta informação está associada com um lugar específico da memória (isso é feito pelo compilador). O **nome** da variável e o endereço da memória onde a informação está armazenada estão associados. O nome e o endereço não mudam. Mas, o valor da informação pode mudar (o valor do que está dentro da caixa pode mudar, embora o tipo seja sempre o mesmo). Cada variável tem um tipo associado. Alguns tipos de variáveis que discutiremos incluem int, string, boolean e float.

Cada variável usa uma determinada quantidade de bytes para seu armazenamento em memória. A maneira como sabemos quantos bytes são utilizados é pelo tipo da variável. Variáveis do mesmo tipo utilizam o mesmo número de bytes, não interessando qual o valor que a variável armazena.

Um dos tipos utilizados para armazenar números é o int. Ele é usado para indicar valores numéricos inteiros.

Textos e caracteres são representados pelo tipo string ou str. Um caracter é um símbolo gráfico (uma letra do alfabeto, um dígito, um símbolo de pontuação, etc) e em um computador é representado por um valor inteiro entre 0 e 255, sendo portanto armazenado em 1 byte de memória. A correspondência entre o valor inteiro e o caracter (texto) que é visualizado em uma tela ou papel é dado por uma tabela internacionalmente aceita chamada Tabela ASCII. O compilador Python faz a tradução para você, portanto você não precisa saber estes números e a correspondência. Um caracter é representado por uma letra, dígito ou símbolo entre apóstrofes (’). Por exemplo, ’C’, ’a’, ’5’, ’$’. Note que ’5’ é um caracter, e não o inteiro 5.

A figura 2.6 mostra como um int e um char são armazenados na memória.

Finalmente, números reais (números com o ponto decimal) são representados pelo tipo float. Este números são armazenados em duas partes: a *mantissa* e o *expoente*. Eles são armazenados de uma maneira que se assemelha a notação exponencial. Por exemplo, o número 6.023×10^{23} é escrito como $6.023e23$. Neste caso, a mantissa é 6.023 e o expoente da base 10 é 23.

Estes números são armazenados de uma forma padrão, tal que a mantissa tem apenas um dígito para a esquerda do ponto decimal. Desta forma, 3634.1 é representado como 3.6341e3, e

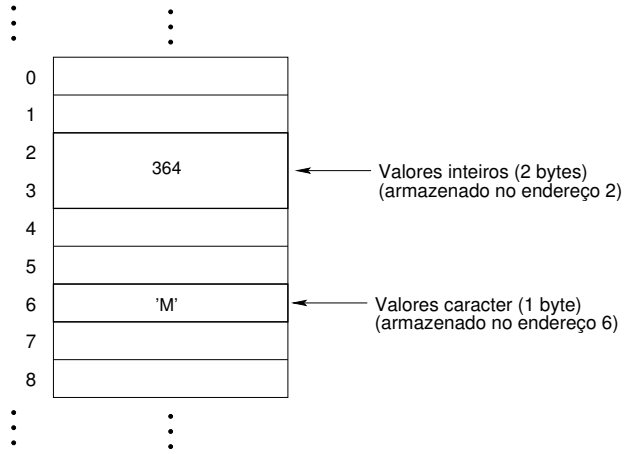


Figura 2.6: Inteiros e caracteres em memória

0.0000341 é representado 3.41e-5.

Definição (criação) de uma Variável

O exemplo `escreve.py` (Figura 2.7) mostra como variáveis são usadas em um programa escrito em Python :

```
#! /usr/bin/env python3
# coding=latin-1

from string import *
from math import *


v = 2.0
print (v)
```

Figura 2.7: `escreve.py`

- `v` é o *nome* da variável.
- a sentença `v = 2.0` representa uma operação de atribuição: o valor expresso no lado direito do sinal `=` é *atribuido* (armazenado) à variável indicada no lado direito. A primeira atribuição de valor a uma variável reserva um endereço de memória para uma variável e o associa a um nome e indica um valor que será armazenado inicialmente no endereço de memória associado. Dizemos então que neste momento ocorreu a *definição da variável*;
- Em Python , o tipo de uma variável é igual ao tipo do valor que é atribuído a ela. O valor `2.0` é um valor real e portanto, o tipo da variável `v` é `float`.

Se você usa variáveis no programa, você deve antes de mais nada defini-las. Isto envolve especificar o nome da variável e atribuir a ela um valor de um determinado tipo, que passará a ser o tipo da variável.

As regras para formar nomes de variáveis em Python são:

- qualquer sequência de letras, digitos, e '_', MAS DEVE COMEÇAR com uma letra ou com '_'. Por exemplo, hora_inicio, tempo, var1 são nomes de variáveis válidos, enquanto 3horas, total$ e azul-claro não são nomes válidos;

- Maiúsculas ≠ Minúsculas;

- Não são permitidos nomes ou palavras reservadas da linguagem:

and	assert	break	class	continue	def
del	elif	else	except	exec	finally
for	from	global	if	import	in
is	lambda	not	or	pass	print
raise	return	try	while	yield	

Tabela 2.1: Palavras Reservadas da Linguagem Python

É sempre uma boa idéia ter certas regras (para você mesmo) para nomear variáveis para tornar o programa mais legível:

- Dê nomes significativos as variáveis (mas não muito longos);

- Use nomes de variáveis do tipo i, j, k somente para variáveis cujo objetivo é fazer contagem;

- Pode-se usar letras maiúsculas ou '_' para juntar palavras. Por exemplo, horaInicio ou hora_inicio. Use o que você preferir, mas SEJA **CONSISTENTE** em sua escolha.

Os tipos básicos de dados existentes em Python são:

Tipo de Dado	Exemplos de Valores
int	10, 100, -786, 080 (octal), 0x2fc (hexa)
long	5192436L, 0122L (octal), 535633629843L
bool	True ou False (com T e F maiúsculos)
complex	3.14j, 3.1 + 4.3j, 3.2+3j
float	-3.34, 2.3e3, 32.3+e18, 32.5-e12
str (*string*)	'eu sou o máximo', 'a'

Em `tipovars.py` (Figura 2.8) temos um exemplo de programa com diversas definições de variáveis:

```
#! /usr/bin/env python3
# coding=latin-1

from string import *
from math import *

pera = 3
qualidade = 'A'
peso = 0.653
flag = true

⋮
```

Figura 2.8: tipovars

Uma forma de determinar os valores de variáveis usando o *operador de atribuição* (=). Colocar um *valor* numa *variável* é chamado de *atribuição*. A notação em Python para atribuição é **=**.

Desta forma, o programa `escreve.py` (Figura 2.7) vai exibir:

```
2.0
```

Para resumir: quando um programa é executado, **uma variável** é associada com:

- **um tipo:** diz quantos bytes a variável ocupa em memória, e como ela deve ser interpretada. Em Python , o tipo de uma variável pode ser determinado pelo comando `type()`.
- **um nome:** um identificador.
- **um valor:** o conteúdo real dos bytes associados com a variável; o valor da variável depende do tipo da variável; a definição da variável não dá valor a variável; o valor é dado pelo operador de atribuição, ou usando a operação `input`. Nós veremos mais tarde que a operação `input` atribui a uma variável um valor digitado no teclado.
- **um endereço:** o endereço do local da memória associado à variável (geralmente onde o byte menos significativo do valor está armazenado).
- Em Python , variáveis devem ser definidas antes de serem usadas, senão ocorrerá um erro de compilação.
- Devem ser dados valores às variáveis antes que sejam utilizadas no programa pela primeira vez.

Constantes (Literais)

Em Python , além de variáveis, nós podemos usar também números ou caracteres cujos valores não mudam. Eles são chamados de constantes. Constantes não são associadas a lugares na memória.

Assim como variáveis, constantes também têm tipos. Uma constante pode ser do tipo int, str, float, etc. Você não tem que declarar constantes, e pode utilizá-las diretamente (o compilador reconhece o tipo pela maneira que são escritos). Por exemplo, 2 é do tipo int, 2.0 é do tipo float, e 'armando' ou '5' são do tipo str.

2.6 Entrada e Saída

Se quisermos que um programa Python mostre alguns resultados, ou se quisermos que o programa peça ao usuário que entre com alguma informação, nós normalmente usamos as funções print() (saída ou exibição de valores na tela) e input() (entrada de dados pelo teclado).

Saída

Para o comando print() fazer o que deve, nós devemos especificar o que será exibido na tela do computador. Nós devemos dar ao comando o que chamamos de *argumentos*. Nos exemplos `hello.py` (Figura 2.5) e `escreve.py` (Figura 2.7) acima, Alô todo mundo! e v são argumentos para print().

Os argumentos de print() podem ser uma variável, uma expressão ou um texto (uma sequência de caracteres entre apóstrofes (')), ou uma combinação dos três.

O comando print() imprime os seus argumentos na tela, seguida de uma mudança de linha (como se o usuário tivesse digitado um ENTER).

O programa `preco.py` (Figura 2.9) mostra um uso básico do comando print().

```
#! /usr/bin/env python3
# coding=latin-1

from string import *
from math import *

PRECO=1.97
pera = 3
qualidade = 'A'
peso = 2.55

print ('Existem', pera, 'peras do tipo', qualidade, '\n com', peso, 'kg.')
print ('O preco por quilo eh R$', PRECO, 'e o total eh R$', peso * PRECO)
```

Figura 2.9: `preco.py`

Observe que:

- A vírgula separa os diferentes argumentos que se quer exibir em uma mesma linha. Na exibição, é colocado automaticamente um espaço em branco entre os valores mostrados.
- Quando se exibe valores do tipo `float`, pode acontecer de serem exibidas casas decimais em excesso, tornando a apresentação do valor pouco confortável. Na aula 5 veremos como configurar a apresentação de valores reais e outros tipos de valores.

- Nós também podemos colocar sequências *escape* dentro de um texto (veja no exemplo a sequência \n antes da palavra pesando).

Entrada

O próximo passo é fazer um programa que lê um conjunto de valores informado pelo usuário e o utiliza para algum processamento.

Em Python a maneira mais simples de ler alguma coisa é utilizando a primitiva input().

Considere o seguinte programa idade.py (Figura 2.10):

```
#! /usr/bin/env python3
# coding=latin-1

from string import *
from math import *

idade = input ('Entre sua idade: ')

print ('Você tem', idade, 'anos')
```

Figura 2.10: idade.py

Este programa mostrará no monitor a mensagem Entre sua idade: (argumento do comando input() e aguardará que um número seja digitado, seguido da tecla ENTER. Depois disso, a variável idade receberá como valor um texto digitado que representa um valor numérico (mas que ainda é apenas um texto. Não é possível fazer operações aritméticas com este valor).

Somente uma *string* pode pode ser lida por um input(). Considere o seguinte exemplo idade-2.py (Figura 2.11):

```
#! /usr/bin/env python3
# coding=latin-1

from string import *
from math import *

dia = input('Entre com o dia de seu aniversario: ')
mes = input('Entre com o mês de seu aniversario: ')
ano = input('Entre com o ano de seu aniversario: ')

print ('Você nasceu em',dia,'/',mes,'/',ano)
```

Figura 2.11: idade-2.py

Para ler os três valores separadamente, precisamos efetuar 3 input()'s. O que significa que o usuário deve digitar 3 valores, cada um seguido por um ENTER. Mais adiante aos estudarmos

strings na aula 5 veremos como fazer para digitar os valores todos em uma única linha, separados por espaço.

input() sempre devolve como valor um texto. Se este texto representa um valor que deve ser usado em cálculos aritméticos, o programador deve antes converter o que foi devolvido por input() para um valor numérico efetivo que o computador possa usar para efeturas cálculos. Expressões aritméticas e como converter textos numéricos em valores numéricos para serem usados nestas expressões é o que será visto na próxima aula.

2.7 Exercícios

1. Instalar em seu computador pessoal o ambiente de programação Python IDEpara uso na aula 3 , seguindo com atenção as instruções para instalação do ambiente .
2. Esboçar os algoritmos (usando Python) que resolvem os problemas abaixo:
   (a) Mostrar a soma de dois números informados pelo usuário;
   (b) Mostrar a média aritmética de três números informados pelo usuário;
   (c) Calcular a raiz de uma equação de 2º grau, cujos coeficientes são informados pelo usuário.

Aula 3

3.1 Introdução

O objetivo desta aula é iniciar o uso do ambiente de programação disponível para Python , compilando alguns dos programas vistos na aula 2 . Em seguida, será abordado o conceito de operações aritméticas, seguindo-se de exercícios práticos.

3.2 Compilando os primeiros programas

Este momento da aula é destinada a ensinar os alunos a instalar o ambiente de compilação, e em seguida a compilar pequenos exemplos de programas em Python .

O aluno deve editar, compilar e executar os programas apresentados pelo professor.

3.3 Expressões Aritméticas

Para que tenha qualquer utilidade, um programa deve ler dados informados pelo mundo externo, processar os valores lidos e apresentar os resultados para o mundo externo de alguma forma percepitível pelo ser humano que está usando o programa.

Vimos até agora uma forma de ler valores do mundo externo (digitação via teclado), armazená-los em variáveis e exibí-los para o mundo externo (tela do computador). O que veremos a seguir são as primeiras formas de especificar operações sobre os valores de variáveis.

Operações Aritméticas

Em Python , nós podemos executar operações aritméticas usando variáveis e constantes. As operações mais comuns são:

`+` adição

`-` subtração

`*` multiplicação

`/` divisão (resultado sempre do tipo `float`)

`%` resto da divisão

`//` quociente inteiro da divisão (*floor division*)

`**` exponenciação / potenciação

Estas operações podem ser usadas em **expressões aritméticas**, como mostram os exemplos abaixo:

```
massa = int(input())
aceleracao = int(input())
fahrenheit = int(input())
```

```
i = 0

celsius = (fahrenheit - 32) * 5.0 / 9.0
forca =  massa * aceleracao
i = i + 1
```

Precedência de Operadores

Em Python , assim como em álgebra, há uma ordem de precedência de operadores.

Assim, em $(2 + x)(3x^2 + 1)$, expressões em parêntesis são avaliadas primeiro, seguidos por exponenciação, multiplicação, divisão, adição e subtração.

Da mesma forma, em Python , expressões entre parêntesis são executadas primeiro, seguidas de exponenciação (`**`), seguidas de `*`, `/` e `%` (que tem todos a mesma precedência), seguido de `+` e `-` (ambos com a mesma precedência).

Quando operações adjacentes têm a mesma precedência, elas são associadas da esquerda para a direita. Assim, `a * b / c * d % e` é o mesmo que `((((a * b) / c) * d) % e)`.

A Operação de Resto (%)

Esta operação é usada quando queremos encontrar o resto da divisão de dois inteiros. Por exemplo, 22 dividido por 5 é 4, com resto 2 ($4 \times 5 + 2 = 22$).

Em Python , a expressão `22 % 5` terá valor `2`.

Note que `%` só pode ser utilizados entre dois inteiros. Usando ele com um operando do tipo `float` causa um erro de compilação (como em `22.3 % 5`).

Expressões e Variáveis

Expressões aritméticas podem ser usadas na maior parte dos lugares em que uma variável pode ser usada.

O exemplo seguinte é válido:

```
raio = 3 * 5 + 1

print ("circunferencia = {0}".format(2 * 3.14 * raio)
```

Exemplos de lugares onde uma expressão aritmética NÃO pode ser usada:

```
yucky + 2 = 5

opps * 5 = input()
```

Este exemplo é ilegal e causará erro de compilação.

Entrada - texto numérico X valor numérico

Vimos ao final da aula 2 que o comando input() devolve um texto como resultado. Se este texto contém dígitos numéricos que representam valores com os quais se deseja fazer cálculos aritméticos diversos, é preciso converter a representação textual para a representação interna do computador, esta sim usada pela CPU para efetuar cálculos.

Considere o programa `expressao.py` (Figura 3.1).

```
#! /usr/bin/env python3
# coding=latin-1

from string import *
from math import *

v = input ()

v = float(v)

v = v * 37 / 2

print ('Valor =', v)
```

Figura 3.1: `expressao.py`

O valor para a variável `v` é lido com input() e ANTES de efetuar o cálculo ($v = v * 37/2$) o TEXTO em `v` é CONVERTIDO para o tipo `float`. Neste momento, a variável deixa de conter um texto e passa a conter um VALOR NUMÉRICO do tipo `float` (real).

OBS: O que acontece se o usuário digitar caracteres não-numéricos? O que é exibido na tela? E se tirarmos a sentença de conversão e digitarmos valores numéricos, o que é exibido na tela?

As funções de conversão existentes em Python são:

int()	texto para inteiro
float()	texto para número real
str()	valor numérico para texto correspondente
chr()	valor ASCII para caracter correspondente
ord()	caracter (texto) para valor ASCII correspondente
complex()	texto para número complexo
bool()	texto para valor booleano

Tabela 3.1: Operações de Conversão de tipos de dados

Expressões envolvendo o operador de atribuição (=)

O formato do operador de atribuição é:

$$lvalue = expressao \tag{2.1}$$

Um *lvalue* (do inglês "left-hand-side value" - valor a esquerda) é um valor que se refere a um endereço na memória do computador. Até agora, o único "lvalue" válido visto no curso é o nome de uma variável. A maneira que a atribuição funciona é a seguinte: a expressão do lado direito é avaliada, e o valor é copiado para o endereço da memória associada ao "lvalue". O tipo do objeto do "lvalue" determina como o valor da *expressao* é armazenada na memória.

Expressões de atribuição, assim como expressões, têm valor. O valor de uma expressão de atribuição é dado pelo valor da expressão do lado direito do `=`. Por exemplo:

`x = 3`	tem valor `3`;
`x = y+1`	tem o valor da expressão `y+1`.

Como consequência do fato que atribuições serem expressões que são associadas da direita para esquerda, podemos escrever sentenças como:

```
i = j = k = 0
```

Que, usando parênteses, é equivalente a `i = (j = (k = 0))`. Ou seja, primeiro o valor 0 é atribuído a `k`, o valor de `k = 0` (que é zero) é atribuído a `j` e o valor de `j = (k = 0)` (que também é zero) é atribuído a `i`.

Operadores de atribuição aritmética

Como foi discutido em classe, estes comandos de atribuição funcionam de forma similar que o comando de atribuição. O lado esquerdo da expressão deve ser um *lvalue*. O valor da expressão de atribuição aritmética é igual ao valor da sentença de atribuição correspondente. Por exemplo:

`x += 3` é igual a `x = x + 3` e tem valor `x + 3`

`x *= y + 1` é igual a `x = x * (y + 1)` e tem valor `x * (y + 1)`

3.4 Exemplos de fixação

Exemplo 2: O que é impresso pelos dois programas abaixo?[1]

[1] `ord()` calcula o valor numérico inteiro da tabela ASCII correspondente ao literal indicado como argumento

```
# coding=latin-1

from string import *
from math import *

score = 5

print (5 + 10 * score % 6) # 7
print (10 / 4) # 2.5
print ( int(10 / 4) ) # 2
print (10.0 / 4.0) # 2.5
print (ord('A') + 1) # B
```

Figura 3.2: Expressoes-1

```
# coding=latin-1

from string import *
from math import *

n1 = input()
n1 = int(n1)
n1 += n1 * 10
n2 = n1 / 5
n3 = n2 % 5 * 7
print (n1, n2, n3) # 55 11 7 (supondo que usuário digitou o número 5)
```

Figura 3.3: Expressoes-2

Exemplo 2: Escreva um programa que leia um número inteiro e imprima `0` (zero) se o número for par e `1` (um) se o número for ímpar.

```
#! /usr/bin/env python3
# coding=latin-1

from string import *
from math import *

numero = input('Entre com um número inteiro: ')
numero = int(numero)
print ('Par? ', numero % 2)
```

Figura 3.4: parimpar.py

Exemplo 3: escreva um programa que obtenha do usuário 3 números inteiros, calcule e exiba a soma, a média aritmética, e produto dos valores lidos.

```
#! /usr/bin/env python3
# coding=latin-1

from string import *
from math import *

n1 = int( input('Entre com um número inteiro: ') )
n2 = int( input('Entre com outro número inteiro: ') )
n3 = int( input('Entre com o último número inteiro: ') )

soma = n1 + n2 + n3;
print('Soma =', soma)
print('Produto =', n1 * n2 * n3)
print('Media =',soma / 3)
```

Figura 3.5: `calcgeral.py`

Observe nos programas `parimpar.py` (Figura 3.4) e `calcgeral.py` (Figura 3.5) que o valor devolvido por `input()` é um texto, que deve ser convertido para um valor numérico efetivo ANTES que se faça qualquer operação aritmética. No primeiro exemplo a conversão é feita em duas sentenças (uma para a entrada de dados e outra para a conversão) e no segundo exemplo a conversão é feita em uma única sentença, usando o valor de `input()` diretamente como argumento de `int()`.

3.5 Exercícios de fixação

1. **(media)** Faça um programa para calcular a média aritmética de 4 valores reais informados pelo usuario.
2. **(idadeAnos2dias)** Faça um programa que obtenha do usuário a sua idade expressa em anos, meses e dias e mostre-a expressa apenas em dias. Considere que 1 ano sempre tem 365 dias e que 1 mês sempre tem 30 dias.
3. **(idadeDias2anos)** Faça um programa que obtenha do usuário a sua idade expressa em dias, e indique na tela a sua idade em anos, meses e dias.Considere que 1 ano sempre tem 365 dias e que 1 mês sempre tem 30 dias.
4. **(segundos2horas)** Faça programa que leia o tempo de duração de um exame expresso em segundos e mostre este tempo expresso em horas, minutos e segundos.
5. **(convFarenheit)** Fazer um programa que lê um número representando uma temperatura na escala Celsius e escreve a correspondente temperatura na escala Farenheit.[2]
6. **(mediaPond)** Faça um programa que leia 3 notas de um aluno e calcule a média final deste aluno. Considerar que a média é ponderada e que o peso das notas é 2, 3 e 5 respectivamente.

[2] $C = \frac{5}{9} \times (F - 32)$

3.6 Exercícios

1. **(convKm)** Fazer um programa que lê um número representando uma velocidade em km/h e escreve a correspondente velocidade em ml/h.[3].

2. **(eqGrau1)** Fazer um programa que lê 2 números, $a \neq 0$ e b e escreve a solução da equação $ax = b$.[4]

3. **(eqSist)** Fazer um programa que lê 6 números, $a_{1,1}$, $a_{1,2}$, b_1, $a_{2,1}$, $a_{2,2}$ e b_2 e escreve uma solução do sistema de equações[5]

$$\begin{aligned} a_{1,1}x + a_{1,2}y &= b_1 \\ a_{2,1}x + a_{2,2}y &= b_2 \end{aligned}$$

[3] 1 milha = 1.609344 km

[4] Observe que o exercício **convKm** é caso particular deste.

[5] Observe que ANTES se faz a dedução analítica para definir expressões aritméticas para o cálculo de x e y, para depois fazer o programa que lê os valores dos coeficientes e calcula x e y.

Aula 4

Muitas vezes existem operações mais complexas para as quais não existe um sinal gráfico específico ou cujo cálculo é complexo demais para existir uma instrução específica correspondente na CPU.

Neste caso as linguagens de programação de um modo geral, a a linguagem Python em particular, disponibilizam um conjunto de operações pré-definidas denomidadas funções[6].

Existem várias funções pré-definidas em Python : funções matemáticas, funções de manipulação de textos, e funções para abrir janelas e para desenhar figuras, e muitas outras. Cada um destes conjuntos de funções é agrupado em uma biblioteca e por isto dizemos que temos bibliotecas de funções e estas funções dizemos ser funções de biblioteca.

Por exemplo, temos a função sqrt(x) que é uma função da biblioteca math (matemática) e ela tem um argumento: um valor numérico inteiro ou real (float). Ao ser usada em um programa, esta função calcula e devolve o valor da raiz quadrada do seu argumento, como podemos ver em raizquad.py (Figura 4.1).

```
#! /usr/bin/env python3
# coding=latin-1

from math import *
from string import *

x = float( input ('Digite um valor numérico: ') )

raiz = sqrt(x)

print ('Raiz quadrada de', x, 'é', raiz)
```

Figura 4.1: raizquad.py

Observe neste exemplo a linha `from math import *` que indica que este programa irá usar as funções da biblioteca math. Sempre que seu programa usar alguma função de uma biblioteca, uma linha deste tipo referente à biblioteca usada deve ser adicionada nas primeiras linhas de seu programa Python .

A tabela 4.1 mostra as principais funções da biblioteca matemática. Nesta tabela, os argumentos são sempre do tipo float e o valor retornado é tmabém sempre do tipo float.

[6] O termo função se refere à forma com que estas operações são implementadas na linguagem e este mecanismo será abordado em detalhes na aula 21

Função	Descrição
$ceil(x)$	retorna o teto de x, isto é, o menor inteiro que é maior ou igual a x.
$fabs(x)$	retorna o valor absoluto de x.
$factorial(x)$	retorna o fatorial de $x(x!)$. x deve ser um valor inteiro e positivo.
$floor(x)$	retorna o piso de x, isto é, o maior inteiro que é menor ou igual a x.
$exp(x)$	retorna e^x (e é a constante de Neper).
$log(x[, b])$	Com um argumento, retorna o logaritmo natural (base e) de x. Com dois argumentos, retorna o logaritmo de x na base b, calculada como $log(x)/log(b)$.
$log10(x)$	retorna o logaritmo de x na base 10.
$pow(x, y)$	retorna x^y. Em particular, $pow(1.0, x)$ e $pow(x, 0.0)$ sempre retorna 1.0, mesmo quando x é zero.
$sqrt(x)$	retorna $\sqrt{x}$.
$cos(x)$	retorna o cosseno de x, sendo x expresso em radianos.
$sin(x)$	retorna o seno de x, sendo x expresso em radianos.
$tan(x)$	retorna a tangente de x, sendo x expresso em radianos.
$degrees(x)$	converte o ângulo x de radianos para graus.
$radians(x)$	converte o ângulo x de graus para radianos.
pi	A cosntante matemática $\pi = 3.141592\cdots$, na precisão disponível.
e	A cosntante matemática $e = 2.718281\cdots$, na precisão disponível.

Tabela 4.1: Funções da Biblioteca math

A tabela 4.2 mostra as principais funções da biblioteca de geração de números aleatórios. Para usar estas funções, o programa deve ter a linha `from random import *` no início do código-fonte.

Função	Descrição
$seed([a])$	inicializa o gerador de números aleatórios com o valor inteiro a. Se este é omitido, a hora atual do sistema é usada ou outra fonte fornecida implicitamente pelo sistema operacional.
$randrange(start, stop[, step])$	retorna um valor aleatório no intervalo $[start, stop)$, considerando a diferença de *step* entre valores consecutivos.
$random()$	retorna um número em ponto flutuante aleatório no intervalo $[0.0, 1.0)$.
$randint(a, b)$	retorna um inteiro aleatório N tal que $a \leqslant N \leqslant b$.
$choice(seq)$	retorna um elemento aleatório de uma sequência (listas, *strings*, etc.) não-vazia *seq*.

Tabela 4.2: Funções da Biblioteca random

Nas próximas aulas veremos que funções existem nas bibliotecas string e sys, conforme forem sendo introduzidas pela primeira vez.

4.1 Exercícios de fixação

1. **(escada)** Faça um programa que receba a medida do ângulo formado por uma escada apoiada no chão e distância em que a escada está de uma parede, calcule e mostre a altura em que a escada toca a parede.

2. **(calculos)** Faça um programa que receba do usuário um número positivo e diferente de zero, calcule e mostre:

   a. A quadrado do número;

   b. A raiz cúbica do número;

   c. A raiz quadrada do número;

   d. O cubo do número.

3. **(volesfera)** Faça um programa que calcula e mostre o volume de uma esfera cujo diâmetro em metros é informado pelo usuário. Lembre-se que o volume de uma esfera é fornecido pela fórmula $V = \frac{4\pi}{3} \times R^3$.

4.2 Exercícios

1. **(eq2grau)** Fazer um programa que lê 3 números, a, b e c satisfazendo $b^2 > 4ac$ e escreve as soluções da equação $ax^2 + bx + c = 0$.

2. **(distpontos)** Construa um programa que, tendo como dados de entrada dois pontos quaisquer no plano, $P(x_1, y_1)$ e $P(x_2, y_2)$, escreva a distância entre eles. A fórmula que efetua

tal cálculo é:

$d = \sqrt{(x_2 - x_1)^2 \ + \ (y_2 - y_1)^2}$

3. **(degraus)** Cada degrau de uma escada tem X de altura. Faça um programa que receba esta altura em centímetros e a altura em metros que o usuário deseja alcançar subindo a escada, calcule e mostre quantos degraus ele deverá subir para atingir seu objetivo, sem se preocupar com a altura do usuário.

Aula 5

5.1 Caracteres e *strings*

Letras, dígitos e outros símbolos gráficos usados em textos são representados em Python pelo tipo de dados `str` (*string*). Variáveis deste tipo contém valores numéricos que são interpretados como caracteres alfanuméricos e gráficos de acordo com uma tabela mantida pelo sistema operacional denominada Tabela ASCII.

Valores do tipo `str` são representados por letras e dígitos entre apóstrofes (por exemplo, 'A'). Todas as letras, números e símbolos que podem ser impressos são escritos desta forma em Python .

Às vezes precisamos de caracteres que não são visíveis, por exemplo, o caracter de "nova linha", que não tem uma representação gráfica específica com é o caso de letras e dígitos, mas o efeito visual de fazer o cursor mudar para a linha abaixo e ir para o início de linha (1ª caracter à esquerda). Neste caso, usa-se uma *sequência de escape*, que é uma sequência de caracteres que inicia com a barra invertida, seguida de dígitos ou letras. A sequência completa representa apenas 1 caracter efetivo.

Por exemplo, `'\n'` representa o caracter para nova linha. Note que embora use-se dois símbolos, estamos representando apenas um caracter. Se quisermos representar o caracter de barra invertida simplesmente, temos que escrever `'\\'`. Algumas sequências de escape comuns são:

'\n': caracter de mudança de linha

'\r': caracter de retorno de carro (cursor vai para 1ª caracter à esquerda na linha, sem mudar de linha)

'\t': caracter de tabulação (8 espaços seguidos)

'\0': caracter de código ASCII 0

'\ddd': caracter de código ASCII `ddd` (valor numérico expresso em base octal, isto é base 8)

'\0xdddd': caracter de código `hhhh` (valor numérico expresso em base hexadecimal, isto é base 16)

'\\': caracter '\'

Cada caracter literal tem um valor inteiro igual ao seu valor numérico do seu código ASCII. Por exemplo, considere o caracter 'A', que tem código ASCII 65, 'B' que tem código 66, e `'\n'` que tem código 12.

Para obter o valor do código ASCII em um programa Python , usamos a função `ord()`. Nós podemos usar a expressão `ord('A') + 1`. O resultado é o valor 66. Se quisermos saber como é a representação gráfica (visível) de um caracter cujo código ASCII se conhece, usamos a função `chr()`. Por exemplo, a expressão `chr(65)` devolve como valor um texto com apenas 1 caracter: 'A'.

A escrita e leitura de *strings* é feita com `print()` e `input()`.

5.2 Operações sobre *strings*

Índices e Substrings

Duas coisas caracterizam uma *string*. Primeiro, para acessar ou extrair um subconjunto de caracteres da *string*, usa-se o conceito de índice. O índice é um valor numérico inteiro que indica a posição do caracter dentro da *string*.

O primeiro caracter de um *string* possui índice 0 (zero), o segundo caracter índice 1 (um) e assim por diante. Se um *string* possui N caracteres, o índice do último caracter de um *string* é $N - 1$.

Um índice pode ser positivo ou negativo, que significa que o índice é relativo ao início do *string* (positivo) ou ao seu final (negativo). Substrings são extraídos fornecendo o intervalo de início e final separados por dois-pontos (':'). Ambas as posições são opcionais o que significa ou começar do início da *string* ou terminar no final da *string*. É proibido acessar posições que não existem. A Tabela 5.1 mostra exemplos de cada uso de índices. Observe que o caracter da posição final em uma operação de extração de substrings nunca é incluído na extração.

Operação	Resultado
seq = 'GAATTC'	
seq[0]	'G'
seq[-1]	'C'
seq[0:3]	'GAA'
seq[1:3]	'AA'
seq[:3]	'GAA'
seq[3:]	'TTC'
seq[3:6]	'TTC'
seq[3:-1]	'TT'
seq[-3:-1]	'TT'
seq[:]	'GAATTC'
seq[100]	$< erro >$
seq[3:100]	'TTC'
seq[3:2]	''

Tabela 5.1: Uso de índices em *strings*

A segunda característica de uma *string* é que ele não pode ser modificado. Uma vez definido seu valor, não é possível alterar diretamente caracteres em seu interior. Para alterar uma string deve-se sempre criar uma nova string contendo a alteração. Observe o exemplo a seguir:

```
a='armando delgado, o belo'
a[3]='A'
```

Figura 5.1: string1

A operação anterior é inválida e sua execução causa a seguinte mensagem de erro:

```
Traceback (most recent call last):
File "teste.py", line 2, in <module>
   a[3]='A'
TypeError: 'str' object does not support item assignment
```

A operação correta é criar uma nova *string*, contendo pedaços do valor da variável `a` e a nova letra 'A' na posição desejada:

```
a='armando delgado, o belo'
b=a[:3]+'A'+a[4:]
print (b)
```

Figura 5.2: string2

A saída do programa acima é:

```
armAndo delgado, o belo
```

A Figura 5.3 ilustra o conceito de strings e substrings.

0	1	2	3	4	5
G	A	A	T	T	C
-6	-5	-4	-3	-2	-1

`seq[1:4] == seq[-5:-2] == AAT`

Figura 5.3: Índices em *strings*

Manipulação de *strings*

A um *string* existem um conjunto de operações e funções associadas que ajudam entre outras coisas a juntar textos, inserir textos dentro de outro, descobrir quantos caracteres possui o texto, etc. Estas funções estão indicadas na Tabela 5.2, onde s, t, *sub* e r indicam uma variável do tipo `str`, i, f e n indicam variáveis do tipo **int**, e as funções de checagem são funções 'booleanas' que retornam $True$ ou $False$.

Operação, Função	Descrição
$s + t$	concatenação de s e t, nesta ordem.
$s * n$	gera *string* com s repetida n vezes.
$len(s)$	retorna valor numérico inteiro que indica o número de caracteres em s

Operação, Função	Descrição
$min(s)$	retorna o menor caracter de s (de acordo com a Tabela ASCII)
$max(s)$	retorna o maior caracter de s (de acordo com a Tabela ASCII)
$s.count(sub)$	conta ocorrências da string sub na string s.
$s.count(sub, i)$	conta ocorrências da string sub na string $s[i:]$.
$s.count(sub, i, f)$	conta ocorrências da string sub na string $s[i:f]$.
$s.find(sub)$	indica o índice em s da 1ª ocorrência da string sub, a partir do início de s. Se sub não é encontrado, retorna o valor -1.
$s.find(sub, i)$	indica o índice em $s[i:]$ da 1ª ocorrência da string sub. Se sub não é encontrado, retorna o valor -1.
$s.find(sub, i, f)$	indica o índice em $s[i:f]$ da 1ª ocorrência da string sub. Se sub não é encontrado, retorna o valor -1.
$s.rfind(sub)$	indica o índice em s da 1ª ocorrência da string sub, iniciando a busca a partir do final de s. Se sub não é encontrado, retorna o valor -1.
$s.rfind(sub, i)$	indica o índice em s da 1ª ocorrência da string sub, iniciando a busca a partir do final de $s[i:]$. Se sub não é encontrado, retorna o valor -1.
$s.rfind(sub, i, f)$	indica o índice em s da 1ª ocorrência da string sub, iniciando a busca a partir do final de $s[i:f]$. Se sub não é encontrado, retorna o valor -1.
$s.lower()$	gera nova string onde todos os caracteres de s são passados para minúscula.
$s.upper()$	gera nova string onde todos os caracteres de s são passados para MAIÚSCULA.
$s.swapcase()$	gera nova string onde as letras minúsculas de s são passadas para MAIÚCULAS, e vice-versa.
$s.capitalize()$	gera nova string onde o 1º caracter de s é colocado em MAIÚSCULA.
$s.title()$	gera nova string onde o 1º caracter de todas as palavras de s são passados para MAIÚSCULA.
$s.center(width)$	gera nova string de comprimento $width$ contendo a string s centralizada.
$s.ljust(width)$	gera nova string de comprimento $width$ contendo a string s alinhada à esquerda.
$s.rjust(width)$	gera nova string de comprimento $width$ contendo a string s alinhada à direita.
$s.lstrip()$	gera nova string contendo s sem espaços em branco em seu início.
$s.rstrip()$	gera nova string contendo s sem espaços em branco em seu final.
$s.strip()$	gera nova string contendo s sem espaços em ambas extermidades.
$s.replace(old, new)$	gera string contendo a string s onde todas as ocorrências da substring old foram substituidas pela substring new.

Operação, Função	Descrição
$s.replace(old, new, maxrep)$	gera string contendo a string s onde no máximo $maxrep$ ocorrências da substring old foram substituidas pela substring new.
$s \quad <, <=, >, >= \quad t$	checa se s está antes ou depois de t considerando ordem alfabética.
$s \quad <, <= \quad t \quad >, >= \quad r$	checa se r está entre s e t considerando ordem alfabética.
$s \quad ==, !=, \textbf{is}, \textbf{is not} \quad t$	checa se s é igual ou diferente de t
$c \quad \textbf{in}, \textbf{not in} \quad s$	checa se c aparece ou não em s.
$s.isalpha()$	checa se todos os caracteres de s são alfabéticos.
$s.isalnum()$	checa se todos os caracteres de s são alfanuméricos.
$s.isdigit()$	checa se todos os caracteres de s são numéricos.
$s.islower()$	checa se todos os caracteres de s são minúsculos.
$s.isupper()$	checa se todos os caracteres de s são maiúsculos.
$s.isspace()$	checa se todos os caracteres de s são espaços em branco.
$s.istitle()$	checa se todas as palavras de s estão capitalizadas (1ª letra em MAIÚSCULO).
$s.endswith(suffix)$	checa se s termina com o sufixo $suffix$.
$s.endswith(suffix, ini)$	checa se a string $s[ini:]$ termina com o sufixo $suffix$.
$s.endswith(suffix, ini, fim)$	checa se a string $s[ini : fim]$ termina com o sufixo $suffix$.
$s.startswith(prefix)$	checa se s começa com o prefixo $prefix$.
$s.startswith(prefix, ini)$	checa se se a string $s[ini:]$ começa com o prefixo $prefix$.
$s.startswith(prefix, ini, fim)$	checa se se a string $s[ini : fim]$ começa com o prefixo $prefix$.

Tabela 5.2: Operações sobre *strings*

Mais funções e operações sobre strings podem ser encontradas em aqui. O mais importante é que a linha `from string import *` seja incluida no início do código fonte do programa.

5.3 Formatação de *strings*

Se quisermos inserir em um texto um valor numérico qualquer para depois imprimir, podemos criar um texto indicando onde os valores devem ser inseridos e, adicionalmente, definir uma forma de apresentação destes valores.

Isto é feito com a operação `format()` associadas a um *string*. Como primeiro exemplo, considere o programa `preco-01.py` (Figura 5.4):

```
#! /usr/bin/env python3
# coding=latin-1

from string import *
from math import *

PRECO=1.97
pera = 3
qualidade = 'A'
peso = 2.557

print ('Existem {0} peras de qualidade {1} pesando {2:.2f} Kg.'.format (pera,
                                                                        qualidade,
                                                                        peso))

print ('O preco por quilo é R$ {0:.2f}, o total é R$ {1:.3f}.'.format (PRECO,
                                                                      peso * PRECO))
```

Figura 5.4: preco-01.py

A saída do programa será:

```
Existem 3 peras de qualidade A pesando 2.56 Kg.
O preco por quilo é R$ 1.97 , o total é R$ 5.037
```

Nas duas mensagens, a função `format()` recebe os valores a serem formatados e dentro do texto da mensagem as indicações {0}, {1}, {2:.2f}, {0:.2f},e {1:.3f} especificam a aparência na tela dos valores indicados como argumentos de `format()`.

Textos formatados contém campos de substituição delimitados por chaves ({}). Qualquer coisa fora das chaves é um texto literal e é copiado sem mudanças para a saída.

O campo de substituição tem o seguinte formato (elementos entre colchetes [ ] são opcionais):

- campo_substituição ::= {*id_campo*[:*formato_campo*]}
- formato_campo ::= [[*fill*]*align*][*sinal*][*width*][.*precisão*][*tipo*]

Essencialmente, o campo {} começa com um *id_campo* que especifica o argumento de format() que será formatado e inserido naquele ponto do texto. Este identificador pode opcionalmente ser seguido pela informação detalhada de formatação, indicado pelos : seguidos por um formato_campo.

O id_campo é um número que indica qual argumento de format() será inserido no lugar. O valor 0 (zero) se refere ao primeiro argumento de format(), 1 (um) ao segundo argumento, e assim por diante. Considere o exemplo abaixo:

```
print( '{0}, {1}, {2}'.format('a', 'b', 'c') )
print( '{2}, {1}, {0}'.format('a', 'b', 'c') )
print( '{0}{1}{0}'.format('abra', 'cad') ) # índices podem se repetir
```

Figura 5.5: campo

A saída será:

```
a, b, c
c, b, a
abracadabra
```

O *formato_campo* é usado para definir como valores individuais são apresentados na saída, e é composto de vários elementos opcionais.

A opção *sinal* é válida apenas para tipos numéricos e indica como os sinais para números positivos e negativos devem ser impressos, conforme descrito na Tabela 5.3:

Opção	Descrição
'+'	mostra o sinal seja tanto para números positivo quanto para números negativos.
'-'	mostra o sinal apenas para números negativos (default).
espaço	mostra um espaço em branco para números positivos e o sinal '-' (menos) para números negativos.

Tabela 5.3: Opções de sinais

O parâmetro *width* é um número inteiro que define o tamanho mínimo do campo. Se não for especificado, então o tamanho é determinado pelo valor a ser impresso. Por exemplo, considere o programa abaixo:

```
print ('Valor = {0}.'.format(3))
print ('Valor = {0}.'.format(39))
print ('Valor = {0:5}.'.format(3))
print ('Valor = {0:5}.'.format(39))
```

Figura 5.6: width

A saída do programa será:

```
Valor = 3.
Valor = 39.
Valor =     3.
Valor =    39.
```

Observe que a presença do parâmetro *width* (no exemplo, o número 5 após dois pontos (:) faz com que espaços em branco sejam colocados para que o campo onde os valores (3 ou 39) serão impressos tenham como tamanho 5 caracteres.

Se desejarmos colocar outro caracter ao invés de espaço para preencher o tamanho do campo, usamos o caracter *fill*, que pode ser qualquer caracter diferente de { ou } e indica o caracter que será usado para preencher o tamanho total do campo quando o valor a ser impresso for menor do que este tamanho. A presença do caracter *fill* é sinalizada pela presença do caracter *align* de alinhamento que o segue, indicados na Tabela 5.4.

Opção de alinhamento	Descrição
'<'	alinha o campo à esquerda dentro do espaço disponível (default).
'>'	alinha o campo à direita dentro do espaço disponível (default para números).
'^'	centraliza o campo dentro do espaço disponível.

Tabela 5.4: Opções de alinhamento

Por exemplo:

```
print ('Valor = {0:^5}.'.format(3))
print ('Valor = {0:#>5}.'.format(3))
print ('Valor = {0:0>5}.'.format(3))
print ('Valor = {0:0^5}.'.format(3))
print ('Valor = {0:0^}.'.format(3))
```

Figura 5.7: fill, align e width

A saída correspondente do programa será:

```
Valor =   3  .
Valor = ####3.
Valor = 00003.
Valor = 00300.
Valor = 3.
```

Note que se *width* não for definido, o campo terá o mesmo tamanho do valor a ser impresso, e assim as informações de alinhamento (*fill* e *align*) não tem efeito neste caso.

A *precisão* é um número decimal que indica quantos dígitos devem ser mostrados antes e após o ponto decimal no caso de valores em ponto flutuante, conforme o que for indicado em *tipo*. Para tipos não-numéricos, indica o tamanho máximo do campo, isto é, quantos caracteres do valor serão usados e impressos. A precisão não é permitida para valores inteiros.

Note que, no caso de números, *width* inclui o TOTAL de caracteres (parte inteira, caracter de ponto decimal, e parte decimal).

Finalmente, o *tipo* determina como o valor deve ser apresentado na saída, isto é, se será um *string*, um valor inteiro ou em ponto flutuante, usando-se diferentes notações.

A Tabela 5.5 apresenta os tipos para inteiros.

Tipo	Descrição
'b'	O valor é impresso usando notação em base 2 (binária).
'c'	Caracter. Converte o inteiro para o caracter correspondente na Tabela ASCIIantes de imprimir.
'd'	O valor é impresso usando notação em base 10 (decimal).
'o'	O valor é impresso usando notação em base 8 (octal).
'x'	O valor é impresso usando notação em base 16, usando letras minúsculas (hexadecimal).
'X'	O valor é impresso usando notação em base 16, usando letras maiúsculas (hexadecimal).
'n'	O mesmo que 'd', exceto que usa o idioma local do sistema operacional para a notação do valor.

Tabela 5.5: Tipos de formato para Valores Inteiros

A Tabela 5.6 apresenta os tipos para valores em ponto flutuante.

Tipo	Descrição
'e'	Imprime o número em notação científica usando a letra e para indicar o expoente..
'E'	Imprime o número em notação científica usando a letra E para indicar o expoente..
'f'	Mostra o número como um número real em ponto fixo.
'g'	Para uma *precisão* $p \geqslant 1$, arredonda o número para p dígitos significativos e então formata o resultado para ponto fixo ou notação científica, conforme for a magnitude.
'n'	O mesmo que 'd', exceto que usa o idioma local do sistema operacional para a notação do valor.
'%'	Multiplica o numero por 100 e o mostra como tipo 'f', seguido do caracter %..
Sem letra	Similar ao tipo 'g', exceto com pelo menos 1 dígito após o ponto decimal e *precisão* de 12.

Tabela 5.6: Tipos de formato para Valores Reais (ponto flutuante)

Exemplos de Formatação

Acessando argumentos pela posição:

```
print( '{0}, {1}, {2}'.format('a', 'b', 'c') )
print( '{2}, {1}, {0}'.format('a', 'b', 'c') )
print( '{0}{1}{0}'.format('abra', 'cad') ) # índices podem se repetir
```

Figura 5.8: argumentos

Saída:

```
a, b, c
c, b, a
abracadabra
```

Alinhamento e tamanho do campo:

```
print( '{:<30}'.format('alinhado à esquerda') )
print( '{:>30}'.format('alinhado à direita') )
print( '{:^30}'.format('centrado') )
print( '{:*^30}'.format('centrado') ) # usa '*' como caracter fill
```

Figura 5.9: alinhamento

Saída:

```
alinhado à esquerda
alinhado à direita
           centrado
**********centrado**********
```

Especificando sinais nméricos:

```
print( 'Valores: {:+f}; {:+f}'.format(3.14, -3.14) ) # mostra sempre
print( 'Valores: {: f}; {: f}'.format(3.14, -3.14) ) # mostra espaço para
positivos
print( 'Valores: {:-f}; {:-f}'.format(3.14, -3.14) ) # mostra apenas o menos
print( 'Valores: {:f}; {:f}'.format(3.14, -3.14) ) # mostra apenas o menos
```

Figura 5.10: sinais

Saída:

```
Valores: +3.140000; -3.140000
Valores:  3.140000; -3.140000
Valores: 3.140000; -3.140000
Valores: 3.140000; -3.140000
```

Mostrando valores inteiros em diferentes bases:

```
print( 'int: {0:d}; hex: {0:x}; oct: {0:o}; bin: {0:b}'.format(42) )
print( 'int: {0:d}; hex: {0:#X}; oct: {0:#o}; bin: {0:#b}'.format(42) )
```

Figura 5.11: bases

Saída:

```
int: 42;  hex: 2a;  oct: 52;  bin: 101010
int: 42;  hex: 0x2A;  oct: 0o52;  bin: 0b101010
```

Expressando percentagens:

```
pontos = 19
total = 22
print( 'Respostas corretas: {:.2%}'.format(pontos/total) )
```

Figura 5.12: percent

Saída:

```
Respostas corretas: 86.36%
```

Mais informações sobre formatação de strings podem ser encontradas aqui.

5.4 Exercícios de fixação

1. **(tamStr)** Mostrar na tela o comprimento de um texto digitado pelo usuário.
2. **(compStr)** Solicitar do usuário dois textos separados e indicar na tela se os textos são iguais, ou se o primeiro texto digitado está antes do outro considerando ordem alfabética.
3. **(juntaStr)** Solicitar do usuário dois textos separados e mostrá-los na tela concatenados, isto é, os dois textos como um só, sem espaço entre eles.
4. **(primaStr)** Solicitar do usuário um texto qualquer e depois um caracter `c` qualquer. O programa então deve mostrar o índice no texto da primeira ocorrência de `c` no texto, ou `-1`, se `c` não ocorre no texto.

5. **(ultimoStr)** Solicitar do usuário dois textos separados e em seguida mostrar o índice da última ocorrência do segundo texto `t` no primeiro, ou `-1`, se o segundo texto não ocorre no primeiro.

6. **(dna)** Calcular a porcentagem dos nucleotídeos G e C em uma cadeia de DNA informada pelo usuário.

Tópico 3

Estruturas de Controle Condicionais

Aula 6

6.1 Introdução

O objetivo desta aula é apresentar o conceito de desvio condicional. Para isto, também será apresentado o conceito de expressões booleanas, em contra-ponto às expressões aritméticas.

6.2 Expressões Booleanas

Expressões aritméticas geram valores numéricos, sendo no caso de textos, temos operações que permitem manipular os caracteres isoladamente ou em conjunto. Mas há situações em que queremos observar dois ou mais valores (numéricos ou texto) e verificar se uma determinada relação entre eles existe ou não. Nestes casos fazemos uso de **expressões booleanas** que quando calculadas geram como valor final os estado True (verdadeiro) ou False (falso).

Operações Relacionais

Em Python , há operadores que podem ser usados para comparar valores: os operadores relacionais.

Há seis operadores relacionais em Python :

$<$ menor que

$>$ maior que

$<=$ menor ou igual que ($\leq$)

$>=$ maior ou igual que ($\geq$)

== igual a

!= não igual a ($\neq$)

in pertinência de *strings*

is identidade (igualdade) entre *strings*

O resultado destas operações é `False` (correspondendo a *0 (zero)*), ou `True` (correspondendo a *1 (um)*). Valores como esses são chamados valores *booleanos*.

Note também que o operador de igualdade é escrito com "sinais de igual duplo", `==`, não `=`. Tenha cuidado com esta diferença, já que colocar `=` no lugar de `==` causa erro de compilação em Python .

Como exemplo, considere o programa `motorista.py` (Figura 6.1).

```
#! /usr/bin/env python3
# coding=latin-1

from string import *
from math import *

idade = 17
print ('Com {0} anos se pode pode tirar carteira de motorista?'.format(idade))
print ('\t', idade >= 18)

idade = 35
print ('Com {0} anos se pode pode tirar carteira de motorista?'.format(idade))
print ('\t', idade >= 18)
```

Figura 6.1: `motorista.py`

A saída deste programa será:

```
Com 17 anos se pode tirar carteira de motorista?
        False
Com 35 anos se pode tirar carteira de motorista?
        True
```

No primeiro conjunto de `print`'s, `idade` é `17`. Logo, `17 >= 18` é falso, o que gera o valor booleano `False` na expressão `idade >= 18`[1].

Depois disso, `idade` é `35`. Logo, `35 >= 18` é verdadeiro, o que gera o valor booleano `True` na expressão `idade >= 18` existente no segundo conjunto de `print`'s.

Considere agora o programa `achaTexto.py` (Figura 6.2), que procura um texto dentro de outro.

[1] `\t` representa a impressão na tela de uma TABULAÇÃO (avanço na linha equivalente a 8 espaços seguidos).

```
#! /usr/bin/env python3
# coding=latin-1

from string import *
from math import *

seq = input()
texto = input()

print ('O primeiro texto está dentro do segundo?')
print ('\t', seq in texto)
```

Figura 6.2: achaTexto.py

Operadores Lógicos

Expressões relacionais testam uma condição apenas. Quando mais de uma condição precisa ser testada, precisamoster uma forma de indicar se estas condições devem ser verdadeira em conjunto, ou se basta que uma delas seja verdadeira.

Para isto, a linguagem Python , assim como a maioria das linguagens de programação de alto nível suportam *operadores lógicos* que podem ser usados para criar *expressões booleanas* mais complexas, combinando condições simples.

Por exemplo, considere o programa parimpar-2.py (Figura 6.3), que indica se um valor inteiro é par ou não e se é positivo ou não.

```
#! /usr/bin/env python3
# coding=latin-1

from string import *
from math import *

num = int ( input() )

print ('Número é par?')
print ('\t', num % 2 == 0)

print ('Número é positivo?')
print ('\t', num > 0)
```

Figura 6.3: parimpar-2.py

Se quisermos saber se o número digitado é par e também positivo, usamos os operadores lógicos:

Neste caso, somente se as duas expressões relacionais tiverem valor True é que o valor da expressão booleana será True.

Os operadores lógicos em Python são

not NÃO lógico, operação de negação

and E lógico, conjunção

or OU lógico, disjunção

A operação de negação, **not**, pode ser usada da seguinte forma:

not *expressão booleana*: O valor é a negação lógica da expressão dada. Por exemplo:

not `True` é `False`

not `False` é `True`

Os dois operadores **and** e **or** operam sobre duas expressões booleanas e tem o resultado `True` (1) ou `False` (0). Os exemplos abaixo mostram o seu uso:

$a == 0$ **and** $b == 0$ (verdadeiro se ambos $a == 0$ e $b == 0$, portanto se `a` e `b` são ambos `0`)

$a == 0$ **or** $b == 0$ (verdadeiro se pelo menos uma das variáveis, ou `a` ou `b` forem `0`)

Uma expressão usando and é VERDADEIRA somente se ambos operandos forem verdadeiros (não zero).

Uma expressão usando or é FALSA somente se ambos operandos forem falsos (zero).

Verifique na Tabela 6.1 o resultado do uso de operadores lógicos:

$expr_1$	$expr_2$	$expr_1$ **and** $expr_2$	$expr_1$ **or** $expr_2$
$verdadeiro$	$verdadeiro$	$verdadeiro$	$verdadeiro$
$verdadeiro$	$falso$	$falso$	$verdadeiro$
$falso$	$verdadeiro$	$falso$	$verdadeiro$
$falso$	$falso$	$falso$	$falso$

Tabela 6.1: Resultado de uso de Operadores Lógicos

Na utilização de expressões lógicas, as seguintes identidades são úteis. Elas são chamadas de *Lei de De Morgan*:

not`(x` **and** `y)` é equivalente a **not** `x` **or** **not** `y`

e

not`(x` **or** `y)` é equivalente a **not** `x` **and** **not** `y`

Expressões aritméticas, relacionais e lógicas

Expressões usando operadores aritméticos, relacionais e lógicos são avaliados. O valor resultante é um número. Para os operadores relacionais e lógicos, este número pode ser 0 (que significa falso) ou 1 (que significa verdadeiro). Por exemplo:

`3 + 5 * 4 % (2 + 8)`	tem valor 3;
`3 < 5`	tem valor 1;
`x + 1`	tem valor igual ao valor da variável `x` mais um;
`(x < 1) or (x > 4)`	tem valor 1 quando o valor da variável `x` é fora do intervalo [1,4], e 0 quando `x` está dentro do intervalo.

6.3 Precedência de operadores

Operadores aritméticos tem precedência maior que os operadores relacionais. Por exemplo, a expressão `3 + 5 < 6 * 2` é o mesmo que `(3 + 5) < (6 * 2)`.

Uma observação sobre valores booleanos – embora você possa assumir que o valor de uma operação relacional é `0` ou `1` em Python , **qualquer valor diferente de zero é considerado verdadeiro**. Falaremos sobre isso mais tarde durante o curso.

A precedência do operador **not** (negação lógica) é a mais alta (no mesmo nível que o "-" unário). A precedência dos operadores lógicos binários é menor que a dos operadores relacionais, e mais alta que a operação de atribuição. O **and** tem precedência mais alta que o **or**, e ambos associam da esquerda para a direita (como os operadores aritméticos).

Como a precedência dos operadores lógicos é menor que a dos operadores relacionais, não é necessário usar parênteses em expressões como:

```
x >= 3 and x <= 50

x == 1 or x == 2 or x == 3
```

A Tabela 6.2 mostra o quadro completo de precedência de operadores aritméticos, relacionais e lógicos.

6.4 Exemplos de Fixação

O que é impresso pelos dois programas abaixo?

Operador	Associatividade
`()`	esquerda para direita
`not - ` (unários)	direita para esquerda
`** * / %`	esquerda para direita
`+ -`	esquerda para direita
`< <= > >=`	esquerda para direita
`== != in is`	esquerda para direita
`and or`	esquerda para direita
`= += -= *= /= %=`	direita para esquerda

Tabela 6.2: Precedência e associatividade de operadores

```
# coding=latin-1

from sys import *
from string import *
from math import *

score = 5

print (5 + 10 * 5 % 6) # 7
print (score + (score == 0)) # 5
print (score + (score > 0)) # 6
```

Figura 6.4: Condicoes-1

```
# coding=latin-1

from sys import *
from string import *
from math import *

n1 = input('Entre com um número inteiro: ')
n1 = int(n1)
n1 *= n1 * 2.5
n2 = n1 / 2
n3 = n2 / 5 * 7
print (n2, n3, (n2 != n3 + 21))
```

Figura 6.5: Condicoes-2

Como é a seguinte expressão completamente parentizada ?

```
a * b / c + 30 >= 45 + d * 3 + e == 10
```

Figura 6.6: Condicoes-3

A resposta:

```
( ( ((a * b) / c) + 30 ) >= ( 45 + (d * 3) + e) ) == 10
```

6.5 Desvio Condicional: o comando if

Em todos os exemplos que vimos até este momento, sentenças são executadas sequencialmente. Mas frequentemente a ordem sequencial de execução de sentenças precisa ser alterada se certas condições forem satisfeitas durante a execução do programa. Isto é chamado *desvio condicional*.

Todas as linguagens de programação oferecem comandos para o desvio condicional. O mais simples é o comando `if`. Em Python , ele tem o formato mostrado na Figura 6.7.

if *expressao* **:**
conjunto de sentenças que são executadas apenas se *expressao* for True

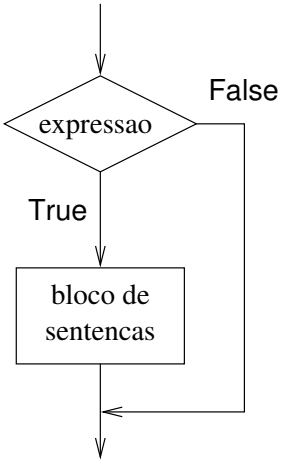


Figura 6.7: O comando `if`

Quando um comando `if` é encontrada em um programa,

1. A *expressao* em parênteses é avaliada.
2. Se o valor de *expressao* for True ou DIFERENTE de zero, as *sentenças* que seguem o '**:**' após *expressao* são executadas em sequência. Este conjunto de sentenças é denominado **bloco**.
3. Se o valor de *expressao* for False ou IGUAL a zero, as sentenças do bloco não são executadas.

4. Deve-se observar que **todas** as linhas do bloco estão **igualmente** *indentadas*, isto é, estão alinhadas a uma mesma distância do caracter 'i' da palavra chave **if**. Esta formatação é obrigatória. Se uma linha não está indentada, ela não faz parte do bloco e portanto não está submetida à expressão do **if**.

5. Avaliado o **if** (com ou sem a execução de seu bloco, o programa prossegue sua execução a partir da sentença seguinte ao bloco, que não estará indentada.

Vejamos agora o exemplo `baskara.py` (Figura 6.8), onde evitamos resolver o problema do que fazer se polinômio tem 2 raízes de igual valor ou se não tem raízes reais.

```
#! /usr/bin/env python3
# coding=latin-1

from math import *
from string import *

a = float( input() )
b = float( input() )
c = float( input() )

delta = b*b - 4*a*c

print ('X1 =', (-b - sqrt(delta))/(2*a) )
print ('X2 =', (-b + sqrt(delta))/(2*a) )
```

Figura 6.8: `baskara.py`

Usando `if` para tratar as diferentes possibilidades, chegamos ao programa `baskara-2.py` (Figura 6.9)

```
#! /usr/bin/env python3
# coding=latin-1

from math import *
from string import *

a = float( input() )
b = float( input() )
c = float( input() )

delta = b*b - 4*a*c

if delta >= 0 :
    print ('X1 =', (-b - sqrt(delta))/(2*a) )

if delta > 0 :
    print ('X2 =', (-b + sqrt(delta))/(2*a) )

if delta < 0 :
    print ('Nao tem raizes reais.')
```

Figura 6.9: baskara-2.py

Considere outro exemplo, que converte uma fração digitada pelo usuário (numerador e denominador) em decimal e imprime o resultado:

```
#! /usr/bin/env python3
# coding=latin-1

from string import *
from math import *


print ('Entre com o numerador e o denominador da fração: ')
a = int ( input() )
b = int ( input() )

print ('A fracao decimal é ',a / b)
```

Figura 6.10: Exemplo-04-0.py

Voce vê algo errado neste programa ? Uma coisa a ser notada é que se o usuário digitar um denominador igual a 0, nós teremos um erro de execução, já que o programa tentaria executar uma divisão por zero. O que é necessário fazer é testar se o denominador é igual a zero e dividir só no caso dele for diferente de zero. Poderíamos reescrever o programa acima da seguinte forma:

```
#! /usr/bin/env python3
# coding=latin-1

from string import *
from math import *


print ('Entre com o numerador e o denominador da fração: ')
a = int ( input() )
b = int ( input() )

if b != 0 :
    print ('A fracao decimal é ',a / b)
```

Figura 6.11: Exemplo-04-1.py

6.6 Exercícios de Fixação

1. **(notas)** Escreva um programa que leia o código de um aluno e suas três notas. Calcule a média ponderada do aluno, considerando que o peso para a maior nota seja 4 e para as duas restantes, 3. Mostre o código do aluno, suas três notas, a média calculada e uma mensagem "APROVADO" se a média for maior ou igual a 5 e "REPROVADO" se a média for menor que 5.

2. **(multiplos)** Elaborar um programa que lê 2 valores a e b e os escreve com a mensagem: "São múltiplos" ou "Não são múltiplos".

6.7 Exercícios

1. **(calcnota)** Escrever um programa que lê o número de identificação, as 3 notas obtidas por um aluno em 3 provas e a média dos exercícios (ME) que fazem parte da avaliação. Calcular a média de aproveitamento, usando a fórmula:

   $MA = (Nota_1 + Nota_2 \times 2 + Nota_3 \times 3 + ME)/7$

   A atribuição de conceitos obedece a tabela abaixo:

Média de Aproveitamento	Conceito
media $>$ 90	A
75 $<$ media $\leqslant$ 90	B
60 $<$ media $\leqslant$ 75	C
40 $<$ media $\leqslant$ 60	D
media $\leqslant$ 40	E

   O programa deve escrever o número do aluno, suas notas, a média dos exercícios, a média de aproveitamento, o conceito correspondente e a mensagem: APROVADO se o conceito for A,B ou C e REPROVADO se o conceito for D ou E.

2. **(medias)** Um usuário deseja um programa onde possa escolher que tipo de média deseja calcular a partir de 3 notas. Faça um programa que leia as notas, a opção escolhida pelo usuário e calcule as médias:

   - aritmética
   - ponderada (pesos 3, 3 e 4)
   - harmônica (definida como sendo o número de termos dividido pela soma dos inversos de cada termo)

3. **(achamaior)** Escreva um programa que leia 3 números inteiros e mostre o maior deles.

4. **(pesoideal)** Tendo como dados de entrada a altura e o sexo de uma pessoa (**M** masculino e **F** feminino), construa um programa que calcule seu peso ideal, utilizando as seguintes fórmulas:

   - para homens: $(72.7 * h) - 58$
   - para mulheres: $(62.1 * h) - 44.7$

5. **(credito)** Um banco concederá um crédito especial aos seus clientes, variável com o saldo médio no último ano. Faça um programa que leia o saldo médio de um cliente e calcule o valor do crédito de acordo com a tabela abaixo. Mostre uma mensagem informando o saldo médio e o valor do crédito.

Saldo médio	Percentual
de 0 a 200	nenhum crédito
de 201 a 400	20% do valor do saldo médio
de 401 a 600	30% do valor do saldo médio
acima de 601	40% do valor do saldo médio

6. **(onibus)** Escreva programa que pergunte a distância em km que um passageiro deseja percorrer e calcule o preço da passagem, cobrando R$ 0,53 por km para viagens de até 200 km e R$ 0,47/km para viagens mais longas.

Aula 7

7.1 `else`

O programa `Exemplo-04-1.py` (Figura 6.11) ficaria ainda melhor se ao invés de não fazer nada no caso do denominador ser zero, imprimirmos uma mensagem de erro ao usuário, explicando o que há de errado.

A sentença em Python que permite fazermos isso é o `if - else`.

O formato do `if-else` é:

if *expressao* **:**
Bloco 1: conjunto de sentenças que são executadas APENAS se *expressao* for True

else :
Bloco 2: conjunto de sentenças que são executadas APENAS se *expressao* for False

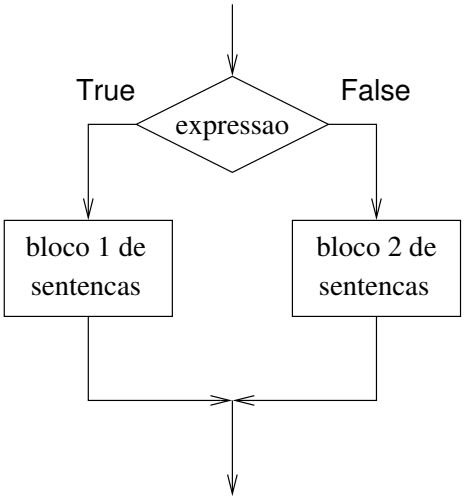


Figura 7.1: O comando `if-else`

Primeiro, a *expressao* é avaliada. Caso a condição seja True (ou diferente de zero), então as sentenças do Bloco 1 são executadas. Caso contrário, as sentenças do Bloco 2 são executadas.

O programa `Exemplo-04-1.py` (Figura 6.11) fica agora:

```
#! /usr/bin/env python3
# coding=latin-1

from string import *
from math import *


print ('Entre com o numerador e o denominador da fração: ')
a = int ( input() )
b = int ( input() )

if b != 0 :
    print ('A fracao decimal é ',a / b)
else :
    print ('Erro: denominador zero!')
```

Figura 7.2: Exemplo-04-3.py

7.2 A construção `if-elif`

Embora ela não seja um tipo diferente de sentença, a seguinte construção é bastante comum para programar decisões entre diversas alternativas excludentes entre si:

if $expressao_1$ **:**

Bloco 1: conjunto de sentenças que são executadas APENAS se $expressao_1$ for True

elif $expressao_2$ **:**

Bloco 2: conjunto de sentenças que são executadas APENAS se $expressao_2$ for True

elif $expressao_3$ **:**

Bloco 3: conjunto de sentenças que são executadas APENAS se $expressao_3$ for True

⋮

else :

Bloco n: conjunto de sentenças que são executadas APENAS se TODAS as expressões anteriores forem False

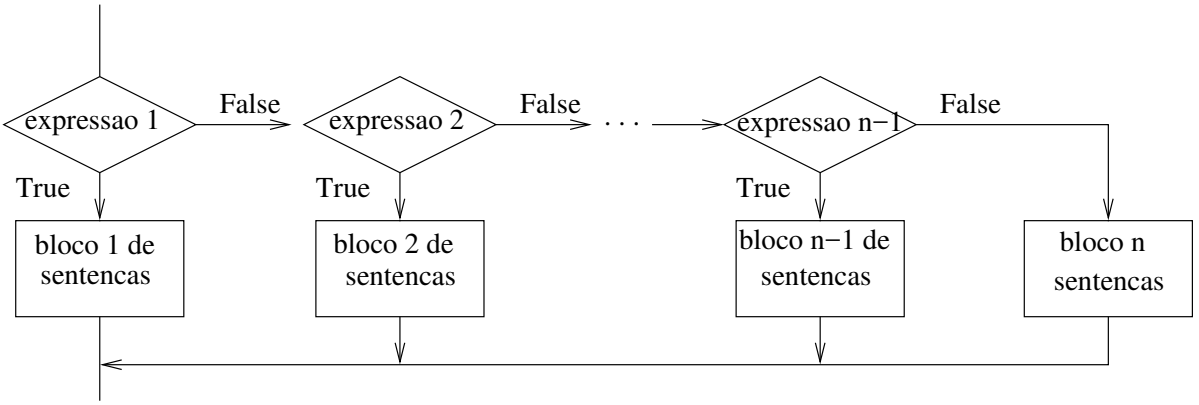


Figura 7.3: O comando `if-elif-else`

As expressões lógicas são avaliadas em ordem, começando com $expressao_1$. Se uma das expressões for verdadeira, as sentenças associadas serão executadas e as demais expressões lógicas não serão avaliadas, e o programa continua após o final da estrutura de `if-elif-else`. Se nenhuma for verdadeira, então a sentença, $sentenca_n$, do último `else` será executada como opção *default*. Se a opção *default* não for necessária, então a parte

else :

conjunto de sentenças que são executadas APENAS se TODAS as expressões anteriores forem False

pode ser removida.

Exemplo 9: O seguinte exemplo mostra um `else-if` de três opções. O programa lê dois números e diz se eles são iguais ou se o primeiro número é menor ou maior que o segundo.

```
#! /usr/bin/env python3
# coding=latin-1

from string import *
from math import *

# obtém 2 números do usuário
num1 = int( input('Entre um número: ') )
num2 = int( input('Entre com outro número: ') )

# mostra a mensagem de comparação
if num1 == num2 :
    print ('Os números são iguais')
elif num1 < num2 :
    print ('O primeiro número é menor')
else :
    print ('O primeiro número é maior')
```

Figura 7.4: Exemplo-04-8.py

No programa acima, se `(num1 == num2)` for verdadeiro, então os números são iguais. Senão, é verificado se `(num1 < num2)`. Se esta condição for verdadeira, então o primeiro número é menor. Se isso não for verdadeiro, então a única opção restante é que o primeiro número é maior.

Exemplo 10: Este programa lê um número, um operador e um segundo número e realiza a operação correspondente entre os operandos dados.

```
#! /usr/bin/env python3
# coding=latin-1

from string import *
from math import *

# obtem uma expressao do usuario
print ('Entre com numero operador numero')
num1 = float (input())
op = input()
num2 = float (input())

print ('{0} {1} {2} = '.format(num1,op,num2), end='')

# mostra o resultado da operacao
if op == '+' :
    print (num1 + num2)
elif op == '-' :
    print (num1 - num2)
elif op == '/' :
    print (num1 / num2)
elif op == '*' :
    print (num1 * num2)
elif op == '%' :
    print (num1 % num2)
elif op == '^' :
    print (num1 ** num2)
else :
    print (' Operador invalido.')
```

Figura 7.5: Exemplo-04-9.py

Exemplos da execução deste programa:

```
Entre com numero operador numero:
5 * 3.5
 = 17.50

Entre com numero operador numero:
10 + 0
 = 10.00

Entre com numero operador numero:
10 x 5.0
 Operador invalido.
```

7.3 Exercícios

1. **(revisao)** Refaça os exercícios da aula 6 , agora usando `else` e `if-elif` onde for adequado.

2. **(calculadora)** Faça programa que leia 2 valores e pergunte qual operação aritmética (+, -, x, /) você deseja efetuar. O programa deve exibir o resultado na tela. Se o usuário indicar operação inexistente, exibir mensagem indicando o erro.

3. **(loja)** Um vendedor precisa de um programa que calcule o preço total devido por um cliente. O programa deve receber o código de um produto e a quantidade comprada. O programa deve calcular e imprimir o preço total usando a tabela abaixo. Além disso, em caso de código inválido uma mensagem deve ser impressa.

código	preço unitário
ABCD	R$ 5,33
XYPK	R$ 6,02
KLMP	R$ 3,27
QRST	R$ 2,59
XXKP	R$ 15,497

4. **(acheseq)** Faça um programa que obtém do usuário duas sequências de nucleotídeos e imprima a frase 'Encontrado' se a segunda sequência for encontrada na primeira. Neste caso o programa deve também indicar em que ponto da primeira sequência a segunda seq. foi encontrada.

7.4 Aninhando senteças `if` e `if-else`

Como era de se esperar, é possível colocar uma sentença condicional dentro de outra. Por exemplo, se quisermos imprimir uma mensagem apropriada caso um número seja positivo ou negativo e par ou ímpar, nós poderíamos escrever o seguinte:

```
#! /usr/bin/env python3
# coding=latin-1

from string import *
from math import *

# Obtem um numero do usuario
num = int(input('Entre com um inteiro: '))

# Imprime uma mensagem dizendo se o numero e positivo ou negativo,

# positivo ou negativo.
if num >= 0 :
    if num % 2 == 0 :
        print ('O numero e par e positivo')
    else :
        print ('O numero e impar e positivo')
else :
    if num % 2 == 0 :
        print ('O numero e par e negativo')
    else :
        print ('O numero e impar e negativo')
```

Figura 7.6: Exemplo-04-5.py

Lembre-se apenas que o `else` está associado a apenas 1 `if`, que é aquele ao qual está alinhado. Não existe `else` sem um `if`, e `else` não tem condição associada (fazer exemplo).

7.5 Exemplos de Fixação

Exercício 1: Assuma as seguintes definições de variáveis:

```
x = 4
y = 8
```

O que é impresso pelos seguintes programas ?

1.
```
if y == 8 :
    if x == 5:
        print ('a ')
    else :
        print ('b ')
print ('c d')
```

2. Altere o programa acima para produzir a seguinte saida:

   - Assuma `x = 5` e `y = 8`

     (a) `a`

(b) `a d`

- Assuma `x = 5` e `y = 7`

  (a) `a c d`

Exercício 2: O que é impresso pelas seguintes sentenças?

1. Assuma `x = 5` e `y = 8`.

```
if x == 5 and y == 8 :
    print ('a')
else :
    print ('b')
```

2. Assuma `x = 4` e `y = 8`.

```
if x == 5 or y == 8 :
    print ('a')
else :
    print ('b')

if not(x == 5 or y == 8) : # equiv. (x != 5 and y != 8)
    print ('a')
else :
    print ('b')


if not(x == 5 and y == 8) : # equiv. (x != 5 or y != 8)
    print ('a')
else :
    print ('b')
```

7.6 Exercícios

1. **(copel)** Escreva programa que calcule preço a pagar pelo fornecimento de energia elétrica. Solicite o consumo em kWh consumidas e o tipo de instalação: R para residencial, I para industrial e C para comercial. Calcule o valor da conta a pagar de acordo com a tabela:

TIPO	Faixa kWh	preço R$
R	até 500	0,40
	acima de 500	0,65
C	até 1000	0,60
	acima de 1000	0,70
I	até 5000	0,50
	acima de 5000	0,60

2. **(acheseq2)** Acrescente ao programa **acheseq** dos exercícios 7.3 a validação dos dados de entrada, isto é, se o usuário informar nucleotídeos inválidos, o programa deve emitir mensagem de erro e terminar sem produzir resultado algum.

3. **(percdna)** Calcular a porcentagem de 2 (dois) nucleotídeos informados pelo usuário em uma cadeia de DNA também informada pelo usuário. O programa deve validar todas as entradas do usuário, isto é, se for informado um nucleotídeo inválido em qualquer uma das entradas do usuário, o programa deve avisar o usuário e encerrar sem fazer cálculo algum.

4. **(abnt)** Escreva programa que leia um nome completo de uma pessoa e imprima na tela o último sobrenome com todas as letras maiúsculas, seguido de vírgula, seguido do restante do nome. Ex.: se entrada for 'Armando Luiz Nicolini Delgado' a saída deve ser 'DELGADO, Armando Luiz Nicolini'

5. **(poluicao)** O departamento que controla o índice de poluição do meio ambiente mantém 3 grupos de indústrias que são altamente poluentes do meio ambiente. O índice de poluição aceitável varia de 0,05 até 0,25. Se o índice sobe para 0,3 as indústrias do 1º grupo são intimadas a suspenderem suas atividades, se o índice cresce para 0,4 as do 1º e 2º grupo são intimadas a suspenderem suas atividades e se o índice atingir 0,5 todos os 3 grupos devem ser notificados a paralisarem suas atividades. Escrever um programa que lê o índice de poluição medido e emite a notificação adequada aos diferentes grupos de empresas.

Tópico 4

Avaliação 1

Aula 8

8.1 Tema da Avaliação 1

Estrutura básica de programas Python , expressões aritméticas e funções de biblioteca matemática. Operações sobre *strings*. Valores booleanos. Expressões relacionais e lógicas. Estrutura condicional IF-ELIF-ELSE.

Solução das questões da prova aqui.

Tópico 5

Estruturas de Controle de Repetição

Aula 9

9.1 Introdução

O objetivo é apresentar a idéia de repetição e o comando `while`, que será o *único* comando de repetição apresentado nestas aulas.

Nesta aula apresentamos inicialmente exemplos triviais para focar a atenção sobre a sintaxe/semântica do `while`.

Progressivamente os exemplos ficam mais elaborados a fim de deslocar a atenção do aluno para a *idéia* de repetição como estrutura elementar do raciocínio algoritmico.

9.2 `while`

A linguagem Python possui comandos para repetir uma sequência de instruções. Estas **estruturas de repetição**, também conhecidas como **laços** (do inglês ***loops***). Estaremos interessados apenas na construção while[1]

Para apresentar o `while`, vamos começar com um exemplo trivial: um programa que lê um número n e escreve os números de 1 a n na tela `seq-01.py` (Figura 9.1).

[1] Outra estrutura de repetiçaõ é o for, que veremos quando tratarmos de listas.

```
#! /usr/bin/env python3
#coding=latin-1

# Imprima todos os números inteiros de 1 a 15.

from string import *
from math import *


i = 1
while  i <= 15 :
    print (i)
    i = i + 1

print('ACABOU !!!')
```

Figura 9.1: seq-01.py

O comando de repetição while tem duas partes: a *expressao* e o *bloco de sentenças da repetição*. O formato do while é mostrado na Figura 9.2.

while *expressao* **:**
conjunto de sentenças que são repetidamente executadas enquanto *expressao* for True

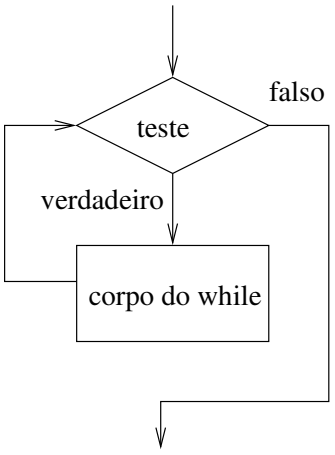


Figura 9.2: O comando `while`

A *expressao* é inicialmente avaliada para verificar se a repetição deve terminar. Caso a expressão seja True (ou diferente de 0 (zero)), o *bloco da repetição* é executado. Depois desta execução, o processo é repetido a partir da avaliação da *expressao*. O *bloco do laço*, por sua vez, pode ser uma única sentença ou várias, da mesma forma que na estrutura if.

No programa seq-01.py (Figura 9.1) a expressão de teste é $i <= N$, e o bloco do laço é composto por 2 sentenças.

Se examinarmos cuidadosamente este exemplo, veremos que a variável i é inicializada com

1 (um) quando é definida. Depois disso, a expressão de teste é verificada e, como 1 <= 15 é verdadeiro, o corpo da repetição é executado. Assim, o programa imprime 1 (valor atual de i), e incrementa i de um. Em seguida, a expressão de teste é verificada novamente e todo o processo se repete até que i seja 15 e 15 seja impresso.

Depois disso, i é incrementado para 16 e o teste é executado. Mas desta vez, 16 ¡ 15 é falso, então o bloco da repetição não é executado e o laço não continua. A execução do programa continua na sentença que segue o laço (no caso, imprimir a frase ACABOU !!!).

Após a execução do while, a variável i tem valor 16.

9.3 Acumuladores

Em seguida, vamos modificar seq-01.py (Figura 9.1) para escrever o valor da soma dos números de 1 a N, onde N será informado pelo usuário somatorio.py (Figura 9.3).

```
#! /usr/bin/env python3
#coding=latin-1

# Calcule e imprima a soma dos inteiros de 1 a N, onde o valor N é informado
# pelo usuário.

from string import *
from math import *


n = int( input() )

soma = 0
i = 1

while  i <= n :
    soma = soma + i
    i = i+1

print (soma)
```

Figura 9.3: somatorio.py

E agora modificamos o programa novamente para escrever o valor do produto dos números de 1 a n produtorio.py (Figura 9.4)

```
#! /usr/bin/env python3
#coding=latin-1

# Calcule e imprima o produto dos inteiros de 1 a N,
# onde o valor N é informado pelo usuário.

from string import *
from math import *


n = int( input('N: ') )

produto = 1
i = 1

while  i <= n :
    produto = produto * i
    i = i+1


print (produto)
```

Figura 9.4: produtorio.py

9.4 Exemplos de Fixação

Exemplo 1: Escrever programa que imprime o valor de x^n sem usar a função pow(x,y)() da biblioteca matemática potencia.py (Figura 9.5).

```
#! /usr/bin/env python3
#!coding=latin-1


x = float( input('Base: ') )
n = float( input('Expoente: ') )


if ( n < 0 ) :
    x = 1/x
    n = -n

r = 1
i = 0
while i < n  :
    i = i+1
    r = r*x

print (r)
```

Figura 9.5: potencia.py

9.5 Exercícios de Fixação

1. **(somaMultiplos)** Calcule e imprima a soma dos inteiros entre 1 a N que são múltiplos de K, onde os valores N e K são informados pelo usuário.

2. **(juntatxt)** Faça um programa que obtenha um conjunto de frases do usuário e as junte em um único texto, na mesma sequência em que foram informadas, com um espaço entre elas. O final da entrada do conjunto inicial de frases ocorre quando o usuário teclar ENTER, sem digitar texto algum. Após juntar as frases, o programa deve imprimir o resultado na tela.

3. **(contapal)** Faça um programa que obtenha um texto do usuário e em seguida solicite palavras do usuário, informando se cada palavra está ou não dentro do texto. O final da entrada das palavras para busca ocorre quando o usuário teclar ENTER, sem digitar texto algum. Após as buscas, o programa deve imprimir o texto inicial e a quantidade de palavras que foram encontradas no texto.

Aula 10

10.1 Entrada de Tamanho Arbitrário

Na aula 9 vimos casos em que a quantidade de vezes que o bloco de sentenças se repete é fixa em um valor específico.

Veremos agora como fazer com que esta quantidade de vezes pode ser dependente de um valor que se informa durante a execução do programa, permitindo que o número de repetições seja arbitrário.

10.2 Exemplos de Fixação

Exemplo 1: Fazer um programa que deve ler uma lista de números e escrever a quantidade de números lidos. O fim da lista é indicado por um 0, que **não** deve ser contado como "número lido" conta.py (Figura 10.1)

```
#! /usr/bin/env python3
#coding=latin-1

# Programa que deve ler  uma lista de números reais
# e imprimir o dobro de cada valor lido.
# Ao final, o programa deve escrever a quantidade de
# números lidos.
# O fim  da lista é indicado por um 0 (zero), que NÃO
# deve ser contado como ‘‘número lido’’.

from string import *
from math import *

qtde = 0
numero = float ( input() )

while numero != 0 :
    print (’** ’, numero * 2)
    qtde = qtde + 1
    numero = float ( input() )


print(qtde)
```

Figura 10.1: conta.py

Exemplo 2: Modifiquemos o programa anterior para um programa que lê uma lista de números terminada por 0 e escreve sua soma. soma.py (Figura 10.2)

```
#! /usr/bin/env python3
#coding=latin-1

# Programa que deve ler  uma lista de números reais
# e calcular a soma de todos eles.
# Ao final, o programa deve escrever a soma e a quantidade
# de números lidos.
# O fim  da lista é indicado por um 0 (zero), que NÃO
# deve ser somado nem contado.

from string import *
from math import *

qtde = 0
numero = float ( input() )
soma = 0

while numero != 0 :
    soma = soma + numero
    qtde = qtde + 1
    numero = float ( input() )

print (soma)
print(qtde)
```

Figura 10.2: soma.py

Exemplo 3: Modifiquemos o programa anterior para um programa que lê uma lista de números terminada por 0 e escreve sua média media.py (Figura 10.3).

```
#! /usr/bin/env python3
#coding=latin-1

# Programa que deve ler  uma lista de números reais
# e calcular a média aritmética de todos eles.
# Ao final, o programa deve escrever na tela o valor
# da média.
# O fim  da lista é indicado por um 0 (zero), que NÃO
# deve ser somado nem contado.

from string import *
from math import *

qtde = 0
numero = float ( input() )
soma = 0

while numero != 0 :
    soma = soma + numero
    qtde = qtde + 1
    numero = float ( input() )

print (soma / qtde)
```

Figura 10.3: media.py

10.3 Exercícios de Fixação

1. **(lanche)** O cardápio de uma lancheria é o seguinte:

Especificação	Código	Preço
Cachorro quente	100	1,20
Bauru simples	101	1,30
Bauru com ovo	102	1,50
Hamburger	103	1,20
Cheeseburguer	104	1,30
Refrigerante	105	1,00

   Escrever um programa que leia o código de um item pedido, a quantidade do item e calcule o valor a ser pago por aquele item. Considere que a cada execução do programa somente será calculado o valor referente a 1 (um) item. Se um valor inválido de código for indicado, o programa deve emitir uma mensagem conveniente e, sem exibir outros resultados, terminar.

2. **(lancheria)** Complete o programa anterior de forma que seja feito o cálculo do preço do lanche até que o usuário entre com código de ítem 0 (zero), quando então o programa deve exibir o valor total e terminar.

3. **(achamaior)** Fazer um programa que seja capaz de ler números inteiros positivos do teclado e de identificar o maior destes números lidos. Ao final, o maior número encontrado

deve ser impresso. O número zero é o ultimo número fornecido como dado de entrada. OBSERVACAO: o zero não deve ser considerado na busca do maior valor.

Aula 11

11.1 Outras estratégias de parada

Esta aula faremos o estudo de outros tipos de repetições em que a condição de parada não envolve um valor digitado ou um limite fixo, mas condições mais diferenciadas (resultado de cálculos, propriedades internas de um texto, etc.).

11.2 Exercícios de Fixação

1. **(achaSpc)** Escreva programa que leia um nome completo de uma pessoa e imprima na tela a localização de todos os espaços existentes no nome. Se não houver espaços, o programa deve indicar isto em uma mensagem conveniente.

2. **(achaPadrao)** Faça um programa que obtenha do usuário 2 sequências de nucleotídeos (denominadas SEQ e PADRAO). O programa deve listar TODAS as posições de PADRAO dentro de SEQ. Caso PADRAO não ocorra em SEQ, o programa deve indicar tal fato com uma mensagem adequada. O programa deve também testar se as sequencias são válidas, isto é, deve testar de nas duas sequências foram indicados apenas nucleotídeos válidos (a, c, t, e g).

3. **(prnabnt)** Escreva programa que leia um nome completo de uma pessoa e imprima na tela o último sobrenome com todas as letras maiúsculas, seguido de vírgula, seguido do restante do nome, onde apenas o 1o. nome deve estar por extenso e o restante abreviado.
   Ex.: se entrada for 'Armando Luiz Nicolini Delgado' a saída deve ser 'DELGADO, Armando L. N.'

4. **(palinTexto)** Faça um programa que descubra se um texto digitado pelo usuário é palíndromo, isto é, se lido da esquerda para a direita ou da direita para a esquerda o texto é o mesmo.

5. **(calcMedia)** Faça programa que obtenha do usuário um conjunto de valores reais e calcule a média aritmética dos valores lidos. O usuário indica o final do conjunto de valores digitando um caracter não numérico diferente de '.'(ponto).

Aula 12

12.1 Repetições Aninhadas

Veremos nesta aula como certas situações em programação implicam em se ter uma repetição cujo bloco de sentenças contém outra repetição, situação esta que chamamos de **repetições aninhadas**.

12.2 Exercícios de Fixação

1. **(conjPadroes)** Faça um programa que obtenha do usuário um conjunto de sequências de nucleotídeos. Para cada sequencia informada, o programa deve solicitar do usuário um conjunto de padroes a serem encontradas. Quando o usuário informar um padrão vazio, o programa pede nova sequencia para análise. Quando o usuário informar uma sequencia vazia, o programa termina. Para cada padrao, o programa deve listar TODAS as posições em que foi encontrado dentro da sequencia em análise. Caso o padrão não ocorra na sequencia, o programa deve indicar tal fato com uma mensagem adequada.

2. **(contaSeq)** Faça programa que leia conjuntos de 2 sequencias de nucleotídeos e, em cada conjunto, indique quantas vezes a 2a. sequencia ocorre na 1a. sequencia. O programa termina quando uma das strings for vazia.

3. ( )

Aula 13

13.1 Exercícios

1. **(nome)**
2. **(nome)**
3. **(nome)**
4. **(nome)**
5. **(nome)**

Tópico 6

Avaliação 2

Aula 14

14.1 Tema da Avaliação 2

Tema da Avaliação 2 (aula 14) acrescido de: Repetições, tratamento mais complexos de textos.

Solução das questões da prova aqui.

Tópico 7

Variáveis Indexadas

Aula 15

15.1 Coleções de Dados

Considere o seguinte programa. Este programa pede ao usuário notas de 7 estudantes, calcula a média e imprime as notas e a média.

```
      int nota0, nota1, nota2, nota3, nota4, nota5, nota6;
      int  media;

      nota0 = int( input( "Entre a nota do estudante 0: ") )
      nota1 = int( input( "Entre a nota do estudante 1: ") )
      nota2 = int( input( "Entre a nota do estudante 2: ") )
      nota3 = int( input( "Entre a nota do estudante 3: ") )
      nota4 = int( input( "Entre a nota do estudante 4: ") )
      nota5 = int( input( "Entre a nota do estudante 5: ") )
      nota6 = int( input( "Entre a nota do estudante 6: ") )

      media = (nota0 + nota1 + nota2 + nota3 + nota4 + nota5 + nota6) / 4;

      print ("Notas:", nota0, nota1, nota2, nota3, nota4, nota5, nota6)
      print ("Media:", media)
  }
```

Este programa é bem simples, mas ele tem um problema. O que acontece se o número de estudantes aumentar ? O programa ficaria muito maior (e feio !!). Imagine o mesmo programa se existissem 100 estudantes.

O que precisamos é uma abstração de dados para agrupar dados relacionados (por exemplo, notas de alunos). Em programação de computadores, temos dois conceitos básicos para resolver este problema:

vetores ou arrays: usados para representar conjunto de valores do mesmo tipo, geralmente relacionados a um mesmo fato ou contexto (por exemplo, notas de alunos, preços de um mesmo produto em lojas diferentes, etc.);

registros: usados para representar conjunto de valores de tipos diferentes, geralmente para caracterizar um único fato ou objeto (por exemplo, dados de uma pessoa em uma agenda, dados de um produto no estoque de uma loja, etc.)

Cada linguagem de programação apresenta construções para representar estes conceitos. De um modo geral, a representação consiste de um conjunto de um ou mais objetos (do mesmo tipo ou de tipos diferentes) armazenados em uma determinada ordem em memória. Cada objeto é chamado de *elemento* da coleção. Da mesma forma que para variáveis simples, damos um nome a este conjunto. O tamanho do conjunto é o seu número de elementos. Cada elemento pode ser identificado de duas formas: através de um número inteiro chamado de *índice*, ou através de um identificador de qualquer tipo (numérico ou textual) denominado *chave*.

Em Python temos 3 formas de representação para os conceitos de vetores e registros:

***string*:** visto na aula 5 , representa uma coleção de valores do tipo caracter, usado para representar textos. Cada caracter é acessado por um índice. O valor do índice começa com `0` (zero) e aumenta de um em um.

lista: representa um conjunto de valores que podem ter tipos diferenciados entre si. Cada valor é acessado por um índice. Como em strings, o valor do índice começa com `0` (zero) e aumenta de um em um. Assim, como acontece com *strings*, o valor do índice do último elemento é igual ao número de valores na lista menos um.

tupla: é similar a uma lista, exceto que é imutável. Uma vez definida uma tupla, não há maneira de mudar seu conteúdo. Fora este detalhe, tuplas são idênticas a listas.

dicionário: como em uma lista, representa conjunto de valores que podem ter tipos diferentes entre si. No entanto, o acesso aos valores é feito não por um índice, mas por uma chave cujo valor pode ser de qualquer tipo.

conjunto (*set*): é similar a uma lista, exceto que elas não são ordenadas e não permitem valores duplicados. Elementos de um conjunto não são acessíveis por um índice (com listas e tuplas) nem por uma chave (como dicionários).

Nesta e nas próximas aulas abordaremos apenas as coleções do tipo lista, sendo que as características de *strings* já foram vistas na aula 5 .

15.2 Definindo Listas e acessando seus elementos

A definição de listas é feita por atribuição (como em variáveis simples), usando-se colchetes `[` e `]` para definir os membros do conjunto, que são separados por vírgula, como mostram os exemplos a seguir:

```
# Definindo uma lista com valores inteiros
nums = [1, 2, 4, 7, 12]

# Definindo uma lista com sequências de nucleotídeos (strings)
pals = ['A', 'CC', 'CGTA', 'ACTGTGCT']

# Definindo  uma lista com  valores de diferentes tipos  (dados de
  uma pessoa)
```

```
pessoa = ['Armando', 16, 100.2, 1.87]

# Definindo uma lista inicialmente vazia (sem valores)
lista = []
```

Para acessar um elemento do lista, devemos indicar a posição do elemento em termos de um índice, que indica a posição a partir do início da lista. Assim, o índice 0 (zero) indica o primeiro elemento da lista, o índice 1 (um) indica o próximo elemento, e assim .por diante.

Em Python isto é indicado da mesma forma que *strings*: ao nome da variável do tipo lista segue-se o índice colocado entre um par de colchetes ([e]). Quando um lista é definida, armazenamento suficiente (bytes contínuos na memória) são alocados para conter todos os elementos do lista.

Exercício de Fixação: Verifique se você entende as sentenças do programa abaixo.

```
nums = [1, 2, 4, 7, 12]
pals = ['A', 'CC', 'CGTA', 'ACTGTGCT']
pessoa = ['Armando', 16, 100.2, 1.87]

x = nums[3]
i = 4
nums[i] = nums[i-1] + nums[i-2]
nums[4] = nums[4] + 1
nums[0] = 2.71828
sala = 3
area = nums[sala] * nums[sala]
pals[2] = input()
```

Agora, podemos reescrever o programa que calcula a média de uma classe de 4 alunos:

```
nota=[0,0,0,0,0,0,0]

indice = 0
while indice < 7 :
   nota[indice] = int( input("Entre a nota do estudante {0}: ".format(indice)))
   indice = indice + 1

print("Notas: ", end='')

total = 0
indice = 0
while indice < 7 :
   print(nota[indice],end=' ')
   total = total + nota[indice]
   indice = indice + 1

print("\nMedia:", total / 7)
```

Exemplo de Saída:

```
Entre a nota do estudante 0: 93
Entre a nota do estudante 1: 85
Entre a nota do estudante 2: 74
Entre a nota do estudante 3: 100
Entre a nota do estudante 3: 56
Entre a nota do estudante 3: 34
Entre a nota do estudante 3: 87
Notas: 93 85 74 100 56 34 87
Media: 75.57
```

O único problema é que ainda não é fácil modificar o programa para tratar uma quantidade diferenciada de alunos porque *7* está em vários pontos do programa. Nós podemos começar com uma lista vazia e ir acrescentando valores usando as funções append() e len(), e a repetição do tipo for[1]:

```
nota = []

num = input("Entre a nota do estudante: ")
while num :
   nota.append(int(num))
   num = input("Entre a nota do estudante ")

print("Notas: ", end='')

total = 0
for n in nota :
   print(n, end=' ')
   total = total + n

print("\nMedia:", total / len(nota))
```

[1]Estas funções e o for serão vistas mais adiante em detalhes

Aula 16

16.1 Operações sobre listas

Índices e Sublistas

Duas coisas caracterizam uma lista. Primeiro, para acessar ou extrair um subconjunto de elementos da lista, usa-se o conceito de índice. O índice é um valor numérico inteiro que indica a posição do elemento dentro da lista.

O primeiro elemento de uma lista possui índice 0 (zero), o segundo elemento índice 1 (um) e assim por diante. Se um lista possui N elementos, o índice do último elemento de um lista é $N - 1$.

Um índice pode ser positivo ou negativo, que significa que o índice é relativo ao início do lista (positivo) ou ao seu final (negativo). Sublistas são extraídas fornecendo o intervalo de início e final separados por dois-pontos (':'). Ambas as posições são opcionais o que significa ou começar do início da lista ou terminar no final da lista. É proibido acessar posições que não existem. A Tabela 16.1 mostra exemplos de cada uso de índices. Observe que o elemento da posição final em uma operação de extração de sublistas nunca é incluído na extração.

Operação	Resultado
seq = [1, 2, 3, 4, 5, 6, 7]	
seq[0]	1
seq[-1]	7
seq[0:3]	[1, 2, 3]
seq[1:3]	[2, 3]
seq[:3]	[1, 2, 3]
seq[3:]	[4, 5, 6, 7]
seq[3:6]	[4, 5, 6]
seq[3:-1]	[4, 5, 6]
seq[-3:-1]	[5, 6]
seq[:]	[1, 2, 3, 4, 5, 6, 7]
seq[::2]	[1, 3, 5, 7]
seq[100]	$< erro >$
seq[3:100]	[4, 5, 6, 7]
seq[3:2]	[ ] ($< listavazia >$)

Tabela 16.1: Uso de índices em *listas*

A segunda característica de uma lista é que ela é mutável, isto é, é possível alterar diretamente elementos em seu interior. Observe o exemplo a seguir:

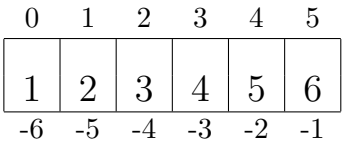
```
a=[1, 2, 3, 4, 5, 6, 7]
a[3]=100
print (a)
```

Figura 16.1: lista1

A operação anterior é válida e sua execução causa a seguinte mensagem:

```
[1, 2, 3, 100, 5, 6, 7]
```

A Figura 16.2 ilustra o conceito de listas e sublistas.




```
seq[1:4] == seq[-5:-2] == [2, 3, 4]
```

Figura 16.2: Índices em *listas*

Aula 17

Manipulação de *listas*

A uma lista existem um conjunto de operações e funções associadas que ajudam entre outras coisas a juntar listas, inserir listas dentro de outra, descobrir quantos elementos possui a lista, etc. Estas funções estão indicadas na Tabela 17.1, onde s e t indicam uma variável do tipo `list`, v indica uma variável de qualquer tipo, i, f, n indicam variáveis do tipo **int**, e as funções de checagem são funções 'booleanas' que retornam $True$ ou $False$.

Operação, Função	Descrição
$s + t$	concatenação de s e t, nesta ordem.
$s * n$	gera lista com s repetida n vezes.
$len(s)$	retorna valor numérico inteiro que indica o número de elementos em s
$del\ \ s[i]$	remove o elemento i da lista s
$min(s)$	retorna o menor elemento de s
$max(s)$	retorna o maior elemento de s
$s.count(v)$	conta ocorrências de v em s.
$s.count(v, i)$	conta ocorrências de v em $s[i:]$.
$s.count(v, i, f)$	conta ocorrências de v em $s[i:f]$.
$s.append(v)$	acrescenta v ao final da lista s (o mesmo que $s = s + [valor]$).
$s.extend(t)$	acrescenta a lista t ao final da lista s (o mesmo que $s = s + t$).
$s.insert(i, v)$	insere $valor$ em s no índice i.
$s.pop()$	recupera o último elemento de s e também o remove de s.
$s.pop(i)$	recupera $s[i]$ e também o remove de s.
$s.remove(v)$	remove o primeiro elemento de s onde $s[i] == valor$.
$s.reverse()$	reverte a posição dos elementos em s.
$s.sort()$	ordena valores de s de forma crescente, ALTERANDO s.
$s.sort(reverse = True)$	ordena valores de s de forma decrescente, ALTERANDO s.
$s.index(v)$	procura v em s, retornando o índice da primeira ocorrência. Caso v não seja encontrado, sinaliza uma excessão ValueError.
$s.index(v, i)$	procura v em $s[i:]$, retornando o índice da primeira ocorrência. Caso v não seja encontrado, sinaliza uma excessão ValueError.
$s.index(v, i, f)$	procura v em $s[i:f]$, retornando o índice da primeira ocorrência. Caso v não seja encontrado, sinaliza uma excessão ValueError.
$v\ \ \textbf{in}, \textbf{not in}\ \ s$	checa se t pertence ou não em s.

Tabela 17.1: Operações sobre *listas*

Por exemplo, considere o seguinte programa:

```
ops = []

valor = input()
while valor :
    ops.append(int(valor))
    valor = input()
```

Neste exemplo, a variável *ops* é uma lista e ela é inicializada vazia na sentença *ops* = [ ]. A repetição é usada para ler um valor pelo teclado e armazená-lo na lista *ops*. Como ele estava inicialmente vazia, usa-se o método *append*() para acrescentar novos valores à lista, sempre na última posição.

Considere agora o programa abaixo:

```
conj = []

# le um par de valores, separados por espaço
valores = input()
while valores :
    pares = valores.split()
    conj.append(pares)
    valores = input()

par = conj[1]
print (par, par[1])
```

Neste exemplo, a variável *conj* é uma lista e ela é inicializada vazia na sentença *conj* = [ ]. O programa recebe em *valores* uma *string* contendo um par de valores separados por espaço. Esta *string* é convertida para uma lista pelo método *split*() na sentença *pares* = *valores.split*(). Assim *pares* é uma lista com 2 elementos. Observe que logo depois, nós armazenamos a lista *pares* em outra lista. Assim, *conj* vai se transformar na verdade em uma lista cujos elementos são por suas vez uma lista também. Após a repetição, a sentença *par* = *conj*[1] cria a lista *par* cujo valor é a lista armazenada no elemento 1 da lista *conj*.

Mais funções e operações sobre listas podem ser encontradas em aqui.

17.1 Gerando Listas a partir de outros dados

As funções indicadas na Tabela 17.2 se referem a operações que envolvem conversão de *strings* em listas e vice-versa. Nesta tabela, considere que *s* e *t* indicam uma variável do tipo `str` e *lst* indica uma variável do tipo `list`.

Operação, Função	Descrição
s.split()	Gera uma lista contendo como elementos as partes de *s* separadas por espaço ou tabulação.
s.split(*sep*)	Gera uma lista contendo como elementos as partes de *s* separadas por *sep*.
s.split(*sep*, *maxsplit*)	Gera uma lista contendo como elementos no máximo *maxsplit* partes de *s* separadas por *sep*.

Operação, Função	**Descrição**
$s.splitlines()$	Quebra s em s em uma lista de linhas.
$s.join(lst)$	junta os elementos da lista lst usando s como separador, produzindo uma nova *string*. Por exemplo, $''.join(lst)$ junta os elementos da lista lst, sem espaço entre eles
$list(s)$	gera uma lista cujos elementos são os caracteres da *string* s.

Tabela 17.2: Operações combinanddo *listas* e *strings*

Aula 18

18.1 Verificação de Limite

Quando uma lista é definida, é alocado espaço em memória para conter todos os elementos do lista (não mais). O tamanho do lista é dado implicitamente, inicializando, acrescentando ou retirando elementos da lista. Embora listas tenham tamanhos específicos, é possível que um programa tente acessar elementos fictícios, isto é, usando valores de índices que não pertencem ao lista. Isto acontece quando usamos um índice que não esteja entre *0* e *n-1* para um lista de tamanho *n*. O compilador não gera nenhum aviso quando isto acontece. Quando executamos um acesso "fora dos limites" do lista, isto causa a interrupção da execução do programa, com a exibição de mensagens tais como as mostradas abaixo:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range

.............

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: <valor> is not in list
```

Por exemplo, considere o seguinte programa:

```
ops = [1,2,3,4,5]

i = 0;
while i <= len(ops) :
    ops[i] = 0
    i = i + 1
```

Este programa deveria inicializar cada elemento do lista com *0*. O problema ocorre quando i tem o valor *5*. Neste ponto, o programa tenta colocar *0* em *ops[5]*. Isto produz uma exceção do tipo IndexError, causando a interrupção do programa após a exibição da mensagem mencionada antes. O problema está na condição do *while*, que não deveria permitir que o corpo da repetição fosse executado quando i assumisse o valor 5. Neste caso, basta simplesmente mudar $<=$ para $<$.

No entanto, este tipo de erro pode ocorrer principalmente quando uma lista tem seu conteúdo aumentado ou diminuído dinamicamente durante o programa, tornando mais complicado ao programador definir o tamanho da lista. Para solucionar o problema, usa-se o tratamento de exceções, tópico que será visto na aula 26 .

18.2 A Estrutura de Repetição FOR

Uma forma de tratar listas em Python sem se preocupar em controlar o valor dos índices é usar uma estrutura de repetição for.

Para apresentar o `for`, vamos começar com um exemplo trivial: um programa que escreve na tela os valores dos elementos de uma lista, um por linha `for-01.py` (Figura 18.1).

```
#! /usr/bin/env python3
#coding=latin-1

# Imprima todos os elementos de uma lista, um por linha

from string import *
from math import *

lista = [2, 4, 5 ,7, 12, 10]

for var in lista :
    print (var)

print('ACABOU !!!')
```

Figura 18.1: `for-01.py`

O comando de repetição for tem três partes: a variável de controle *var*, uma *lista* e o *bloco de sentenças da repetição*. O formato do for é mostrado na Figura 18.2.

for *var* **in** *lista* **:**
conjunto de sentenças que são repetidamente executadas. Em cada iteração, a variável *var* assume o valor do um elemento de *lista*. A repetição termina quando depois que *var* assumir o último valor armazenado em *lista*.

Figura 18.2: O comando `for`

Caso a lista não esteja inicialmente vazia, a variável *var* assume inicialmente o valor do primeiro elemento de *lista*. , o *bloco da repetição* é executado. Depois desta execução, o processo é repetido, sendo que *var* assume o valor do elemento seguinte de *lista*. Se não há mais elementos, a repetição se encerra. O *bloco da repetição*, por sua vez, pode ser uma única sentença ou várias, da mesma forma que na estrutura while ou if. Além disso, a variável *var* pode ser usada em qualquer sentença do bloco.

Observe que neste caso não foi necessário o uso de índices ou de testar os limites de índices, como na Seção 18.1.

Finalmente, o comando for é muito usado com a função range(). Esta função é usada para gerar um conjunto de valores pela indicação de um valor inicial e final, e do intervalo entre os valores. Por exemplo:

range(0,10) gera os numeros de 0 a 9

range(10) gera os numeros de 0 a 9. É equivalente a range(0, 10)

range(5,8) gera os numeros de 5, 6 e 7

range(0, 10, 2) gera os numeros de 0, 2, 4, 6, 8

```
#! /usr/bin/env python3
#coding=latin-1

# Imprima todos os numeros de 3 a 70 que são múltiplos de 3

from string import *
from math import *

for var in range(3,70,3) :
    print (var, end=' ')

print()
```

Figura 18.3: for-02.py

Exercícios

1. **(vetor)** Faça programa que lê os valores inteiros N e $a_1, a_2, \ldots, a_N$, e monta em uma lista `lst` os valores lidos. Em seguida, o programa deve escrever na tela os valores da lista `lst` na forma $(a_1, a_2, \ldots, a_N)$.

2. **(media)** Fazer programa que calcula a média aritmética dos elementos de uma lista. Os valores da lista são informados pelo usuário, que encerra sua entrada de dados ao digitar ENTER sem valor algum.

3. **(ordena)** Faça programa que leia diversos valores inteiros do usuário e os armazene em uma lista de inteiros. Após a leitura, o programa deve calcular a média dos valores armazenados na lista e mostrar na tela a lista ordenada em ordem decrescente e o valor da média. Os valores DEVEM SER digitados em uma única linha, separados por espaço.

4. **(palinLista)** Faça um programa que descubra se um texto digitado pelo usuário é palíndromo, isto é, se lido da esquerda para a direita ou da direita para a esquerda o texto é o mesmo. Use apenas operações com listas e *strings*. NÃO USE REPETIÇÕES.

5. **(soma_listas)** Faça programa que lê duas listas distintas de números inteiros e cria uma terceira lista em que cada elemento é a soma dos valores correspondentes das duas primeiras listas, isto é, o primeiro elemento da lista final é a soma dos primeiros elementos das duas listas iniciais, o segundo elemento da lista final é a soma dos segundos elementos das duas listas iniciais, e assim por diante.

6. **(criaNova)** Faça programa que leia duas listas de inteiros do usuário e gere e imprima um terceira lista com os elementos das duas primeiras, mas sem repetições.

7. **(qtpaa)** Deseja-se fazer um levantamento na população do município de Passaredo para verificar a incidência de pressão arterial alta (PAA) naquela população. Faça um programa que lê do usuário um conjunto de dados que consiste de um par de valores representando a idade de uma pessoa e a quantidade de vezes que ela teve pressão alta no ano passado. A aquisição destes dados termina quando o usuário digitar ENTER sem valor algum.
   De posse destes dados, o programa deve solicitar do usuário uma idade e um valor de referência para número de ocorrências de PAA. De posse destes dois dados, o programa deve finalmente calcular e mostrar na tela a quantidade de pessoas da idade informada que tiveram PAA acima do valor de referência informado.

8. **(mediapaa)** Deseja-se fazer um levantamento na população do município de Passaredo para verificar a incidência de pressão arterial alta (PAA) naquela população. Faça um programa que lê do usuário um conjunto de dados que consiste de um par de valores representando a idade de uma pessoa e a quantidade de vezes que ela teve pressão alta no ano passado. A aquisição destes dados termina quando o usuário digitar ENTER sem valor algum.
   De posse destes dados, o programa deve gerar e exibir uma lista contendo um trio de valores. Cada trio contém um valor de idade (dentre os coletados inicialmente), a quantidade de pessoas com esta idade, e a média de ocorrências de PAA para aquela idade.

9. **(buscapaa)** Complete o exercício anterior, fazendo que o programa peça ao usuário um valor de idade e mostre na tela, a partir da lista de trios, a quantidade de pessoas daquela idade e a média de ocorrências de PAA para aquela idade.

Aula 19

Exercícios

1. **(doenca)** Deseja-se fazer um levantamento na população do município de Passaredo para verificar a incidência de pressão arterial alta (PAA) naquela população. Faça um programa que lê do usuário um conjunto de dados que consiste de um par de valores representando a idade de uma pessoa e a quantidade de vezes que ela teve pressão alta no ano passado. A aquisição destes dados termina quando o usuário digitar ENTER sem valor algum.
   De posse destes dados, o programa deve calcular e mostrar na tela os seguintes resultados: (a) Tabela indicando a média de ocorrências de PAA por idade; (b) A quantidade de pessoas, também por idade, que tiveram PAA no ano passado uma quantidade de vezes acima da média.

2. **(estoque)** Faça um programa que lê do usuário um conjunto de dados do estoque de uma mercearia. Cada item do estoque é definido por: nome do item, quantidade no estoque, e preço unitário do item. A entrada dos dados termina quando o usuário informa um nome de item vazio. Após a entrada de dados, o programa deve permitir que o usuário informe diversas operações de venda da mercearia. O usuário informa o nome do item vendido e a quantidade a ser vendida. O programa não deve permitir a venda de produtos em quantidade não disponível no estoque. A cada venda, a quantidade do item no estoque deve ser devidamente reduzida. Na fase de venda, quando o usuário informar um nome de item vazio, o programa termina exibindo a relação do estoque, e o valor total arrecadado com as vendas, indicando também os dados dos três ítens mais vendidos.

Tópico 8

Avaliação 3

Aula 20

20.1 Tema da Avaliação 3

Tema da Avaliação 2 (aula 14) acrescido de: Variáveis Indexadas.

Solução das questões da prova aqui.

Tópico 9

Funções

Aula 21

21.1 Introdução

Quando queremos resolver um problema, em geral tentamos dividi-lo em subproblemas mais simples e relativamente independentes, e resolvemos os problemas mais simples um a um. As linguagens de programação em geral (a Python em particular) dispõem de construções (abstrações) que auxiliam o projeto de programas de maneira. Uma função cria uma maneira conveniente de encapsular alguns detalhes de "processamento", ou seja, como algum resultado é obtido. Quando esta "computação" é necessária, a função é *chamada*, ou *invocada*. Desta forma, quando uma função é chamada o usuário não precisa se preocupar com a forma com que a computação é realizada. É importante saber o que a função faz (qual o resultado da execução de uma função) e também como se usa a função. Criando funções, um programa Python pode ser estruturado em partes relativamente independentes que correspondem as subdivisões do problema.

Você já viu algumas funções: `pow()`, `input()`, `print()`, `sqrt()`. Elas são funções de bibliotecas padrão em Python . Você não sabe como elas foram escritas, mas já viu como utilizá-las na aula 4 . Ou seja, você sabe o *nome* da função e quais informações específicas você deve fornecer a ela (valores que devem ser *passados* como parâmetros para a função) para que a função produza os resultados esperados.

Tomemos agora como exemplo o programa abaixo, que obtém duas listas de inteiros fornecidas pelo usuário e gera uma terceira lista contendo a união das duas primeiras, sem repetição de valores:

```
#coding=latin-1

# Faça programa que leia duas listas de inteiros do usuário
# e gere e imprima um terceira lista com os elementos das
# duas primeiras, mas sem repetições.
#

# Inicio do programa principal

# Preenche lista 1 com dados do usuario
lis1=[]
print('Preencha a 1a. lista:')
num = int( input('Valor: ') )
while num != 0 :
    lis1.append(num)
    num = int( input('Valor: ') )

# Preenche lista 2 com dados do usuario
lis2=[]
print('Preencha a 2a. lista:')
num = int( input('Valor: ') )
while num != 0 :
    lis2.append(num)
    num = int( input('Valor: ') )

# Cria a nova lista vazia
nlis = []

# Acrescenta lista 1 à nova lista, exceto o que
# estiver repetido.
for val in lis1 :
    if val not in nlis :
        nlis.append(val)

# Acrescenta lista 2 à nova lista, exceto o que
# estiver repetido.
for val in lis2 :
    if val not in nlis :
        nlis.append(val)

# Imprime a lista final
print (nlis)
```

Figura 21.1: uniaoListas.py

Observe que o código que faz a leitura das listas é a mesma, diferenciando-se apenas pela lista que preenche. O mesmo acontece com a formação da lista nova sem repetições.

Um dos benefícios mais óbvios de usar funções é que também podemos evitar *repetição de código*. Em outras palavras, se você quiser executar uma operação mais de uma vez, você pode

simplesmente escrever a função uma vez e utilizá-la diversas vezes ao invés de escrever o mesmo código várias vezes. Outro benefício é que se você desejar alterar ou corrigir alguma coisa mais tarde, é mais fácil alterar em um único lugar.

O exemplo anterior pode ser simplificado pela criação das funções `lerLista`, que obtém um onjunto de valores do usuário e retorna como resultado uma lista com estes valores, e `unirListas`, que recebe duas listas e retorna como resultado uma lista contendo a união das duas listas recebidas, sem repetições:

```
#coding=latin-1

# Função que obtem conjunto de inteiros de usuário e retorna
# uma lista com estes valores
def lerLista() :
    lis=[]
    num = int( input('Valor: ') )
    while num != 0 :
        lis1.append(num)
        num = int( input('Valor: ') )

    return lis

# Função que une duas listas e retorna como resultado uma unica
# lista com a união das duas primeiras, sem repetições
def unirListas(dest, lista) :
    for val in lista :
        if val not in dest :
            dest.append(val)

    return dest
```

Figura 21.2: lerLista.py

Observe o uso da instrução `return`. Ela é usada para indicar qual o resultado final produzido pela função, indicado pelo argumento de `return`, Este argumento pode se uma variável única, ou um conjunto de variáveis, separados por vírgula, quando a função tiver que produzir mais que 1 resultado.

O exemplo uniaoListas.py (Figura 21.1) pode ser então alterado e simplificado com o uso das funções `lerLista()` e `unirListas()`[1]:

[1] Quando nos referirmos a uma função neste texto usaremos a maneira frequentemente utilizada que é o nome da função seguido de `()`.

```
#coding=latin-1

# Faça programa que leia duas listas de inteiros do usuário
# e gere e imprima um terceira lista com os elementos das
# duas primeiras, mas sem repetições.
#

# Função que obtem conjunto de inteiros de usuário e retorna
# uma lista com estes valores
def lerLista() :
    lis=[]
    num = int( input('Valor: ') )
    while num != 0 :
        lis1.append(num)
        num = int( input('Valor: ') )

    return lis

# Função que une duas listas e retorna como resultado uma unica
# lista com a união das duas primeiras, sem repetições
def unirListas(dest, lista) :
    for val in lista :
        if val not in dest :
            dest.append(val)

    return dest

# Inicio do programa principal

# Preenche lista 1 com dados do usuario
print('Preencha a 1a. lista:')
lis1 = lerLista()

# Preenche lista 2 com dados do usuario
print('Preencha a 1a. lista:')
lis2 = lerLista()

# Cria a nova lista vazia
nlis = []

# Acrescenta lista 1 à nova lista, exceto o que
# estiver repetido.
nlis = unirListas(nlis,lis1)

# Acrescenta lista 2 à nova lista, exceto o que
# estiver repetido.
nlis = unirListas(nlis,lis2)

# Imprime a lista final
print (nlis)
```

Figura 21.3: uniaoListas-func.py

Como pode ser observado, sejam quais forem as 2 listas fornecidas, não é necessário escrever um código similar para cada lista (como feito em `uniaoListas.py` (Figura 21.1)). Basta `chamar` a funções `lerLista()` e `unirListas()`, passando os valores necessários para fazer as operações necessárias, e utilizar os resultados retornados pela função.

Evitar repetição de código é a razão histórica que funções foram inventadas (também chamado de procedimento ou subrotinas em outras linguagens de programação). Mas a maior motivação para utilizar funções nas linguagens contemporâneas é a redução da complexidade do programa e melhoria da modularidade do programa. Dividindo o programa em funções, é muito mais fácil projetar, entender e modificar um programa. Por exemplo, obter a entrada do programa, realizar as computações necessárias e apresentar o resultado ao usuário pode ser implementado como diferentes funções chamadas pelo programa principal nesta ordem.

Funções podem ser escritas *independentemente* uma da outra. Isto significa que, em geral, variáveis usadas dentro de funções não são compartilhadas pelas outras funções, tornando qo comportamento da função é previsível. Se não for assim, duas funções completamente não relacionadas podem alterar os dados uma da outra. Se as variáveis são *locais* a uma função, programas grandes passam a ser mais fáceis de serem escritos. A comunicação entre funções passa a ser controlada – elas se comunicam somente através dos valores passados às funções e dos valores retornados.

21.2 Definição de Funções

Além das funções disponíveis na biblioteca, podemos definir as nossas próprias funções num programa Python . Assim, um programa Python na verdade consiste do código principal, possivelmente precedido pela definição de uma ou mais funções, que serão chamadas no código principal ou em outras funções. Cada função pode ser definida separadamente. Abaixo, mostramos um exemplo simples de um programa que consiste de uma função de nome `alo()` e do código principal. Quando executado, este programa imprimirá a mensage `Alo!` três vezes:

```
#coding=latin-1

# definição da função alo()
def alo() :
    print ('Alo!')

# Programa Principal
i = 1
while i <= 3 :
    alo()
    i = i + 1
```

Figura 21.4: `alo.py`

O formato geral da definição de uma função é

def *nome-da-função* **(** *lista-de-argumentos* **) :**

declarações e sentenças

retorno dos resultados

A primeira linha da definição é o cabeçalho da função. Ela têm duas partes principais: o nome da função, e entre parênteses uma lista de *parâmetros* (também chamado de *argumentos formais*). Após este cabeçalho, deve ser definido o corpo de sentenças da função como um *bloco* de sentenças, indentado em relação ao cabeçalho. A função devolde seu resultado através de uma sentença especial. Para simplificar a exposição, falaremos sobre o *retorno* e os *argumentos formais* mais tarde. Eles servem para permitir que as funções troquem informações entre si e com o código principal.

Observe que o código principal (no programa `alo.py` (Figura 21.4), as sentenças após o comentário # Programa Principal) consiste de sentenças fora do corpo/bloco de definição das funções. A execução do programa sempre se iniciará na 1ª sentença do código principal.

Todas as funções devem ser definidas antes de serem usadas. As funções da biblioteca padrão, tais como `sqrt()`, são pré-definidas, mas mesmo assim devem ser declaradas (deve ser anunciado ao compilador que elas existem). É por isso que incluímos a linha `from math import *` no início do código fonte quando usamos funções matemáticas nos programas.

Funções sem argumentos

Para começar, vamos utilizar funções na seguinte forma:

def *nome-da-função*( ) :

declarações e senteças (corpo da função)

Esta é uma função que não gera valores para serem usados posteriormente pelo programa, o que é caracterizado pela ausência de uma instrução para retorno de valores. O par de parênteses vazio significa que a função não tem argumentos (ela não precisa de nenhuma informação externa para ser executada). Isso não significa que a função não faz nada. Ela pode realizar alguma ação, como imprimir uma mensagem. O exemplo `alo.py` (Figura 21.4) mostra um programa que usa uma função como essa.

A função `alo()` imprime a mensagem `Alo!` quando chamada. O corpo da função consiste apenas da sentença `print()`. Dentro do código principal há uma *chamada* a função `alo()`. A função é chamada pelo seu nome seguido de `()` (já que a função `alo` não tem argumentos, nenhuma expressão é escrita dentro dos parênteses). A função `alo()` não retorna um valor, ela é chamada simplesmente para realizar uma ação (imprimir a mensagem).

Funções com argumentos

Nosso próximo exemplo pede que o usuário digite suas iniciais, e então chama a função cumprimenta() para imprimir a mensagem "Ola" junto com as iniciais digitadas. Estas iniciais (seus valores) são *passadas* para a função cumprimenta(). A função cumprimenta() é definida de forma que ela imprimirá a mensagem incluindo quaisquer iniciais passadas para seus *parâmetros formais* inic1 e inic2.

```
#coding=latin-1

def cumprimenta(inic1, inic2) :
    print ('Olá, ', inic1, inic2, '!')

# Final da Função

# Programa Principal

print ('Entre com duas iniciais: ')
primeiro = input()
segundo = input()

cumprimenta(primeiro, segundo)
```

Figura 21.5: cumprimenta.py

O código principal chama a função cumprimenta(), passando para ela os valores dos dois caracteres para serem impressos. Veja um exemplo de execução do programa:

```
Entre com duas iniciais:
Y
K
Alo, YK!
```

Note que há uma correspondência entre a quantidade e a posição dos valores que o programa passa (estes são chamados de *parâmetros reais* ou *argumentos reais*) e os parâmetros formais listados no cabeçalho da função `cumprimenta()`.

Funções que retornam um valor

Uma função também pode *retornar* um valor para o programa que o chamou. O valor retornado pode ser de qualquer tipo. Uma função que retorna um valor executa alguns cálculos, e retorna o resultado para quem a chamou. A função chamadora pode então usar o resultado. Para retornar um valor para a função chamadora, a função usa a sentença `return`.

O formato da sentença `return` é a seguinte:

return *expressão*

A *expressão* é avaliada e o seu valor é devolvido ao ponto de chamada da função.

Considere o programa triReto.py (Figura 21.6). O programa consiste da parte principal e da função `quad()`. O programa pede que o usuário digite três números e verifica se eles podem ser os lados de um triângulo reto.

```
#coding=latin-1
# programa que verifica se 3 numeros podem ser os lados de um
# triangulo reto.
#

# funcao que calcula o quadrado de um numero
def quad(n) :
    return n * n

# Programa Principal

print('Entre tres inteiros: ')
s1 = input()
s2 = input()
s3 = input()

if s1 > 0 and s2 > 0 and s3 > 0 and
  (quad(s1) + quad(s2) == quad(s3) or
   quad(s2) + quad(s3) == quad(s1) or
   quad(s3) + quad(s1) == quad(s2)) :

    print (s1, s2, s3, 'podem formar um triangulo reto')
else :
    print (s1, s2, s3, 'não podem formar um triangulo reto')
```

Figura 21.6: triReto.py

Note que quando chamamos a função `quad()` passamos o valor com o qual desejamos executar o cálculo, e também usamos o valor retornado pela função em expressões. O valor de `quad(s1)` é o valor que a função `quad()` retorna quando chamado com o valor do argumento sendo igual ao valor da variável `s1`.

Os valores retornados pelas chamadas de funções podem ser usados em todos os lugares valores podem ser usados. Por exemplo,

```
y = quad(3)
```

Aqui `quad(3)` tem o valor `9`, portanto `9` será atribuído à variável `y`;

```
x = quad(3) + quad(4)
```

atribuirá `25` à variável `x`, e

```
area = quad(tamanho)
```

atribuirá à variável `area` o valor da variável `tamanho` elevado ao quad.

O exemplo cinco.py (Figura 21.7) tem uma função chamada `cinco`:

```
def cinco() :
    return 5

# Programa Principal
print ( 'cinco = ', cinco() )
```

Figura 21.7: cinco.py

A saída do programa será

```
cinco = 5
```

porque o valor de `cinco()` dentro da sentença `print` é `5`. Olhando na sentença `return`, `5` é a expressão retornada para o chamador.

Outro exemplo:

```
#coding=latin-1

def obtem_valor() :
    valor = int( input('Entre um valor: ') )

    return valor

# Programa Principal

a = obtem_valor()
b = obtem_valor()

print ( 'soma = ', a + b )
```

Figura 21.8: obtem-valor.py

Este programa obtém dois inteiros do usuário e mostra a sua soma. Ele usa a função `obtem_valor()` que mostra uma mensagem e obtém um valor numérico inteiro do usuário.

Um exemplo de saída deste programa é:

```
Entre um valor: 15
Entre um valor: 4
soma = 19
```

Aula 22

22.1 Mais sobre o `return`

Quando `return` é executada, a função imediatamente acaba – mesmo que haja código na função após a sentença `return`. A execução do programa continua após o ponto no qual a chamada de função foi feita. Sentenças `return` podem ocorrer em qualquer lugar na função – não somente no final. Também é válido ter mais de um `return` dentro de uma função.

O seguinte exemplo mostra uma função (uma versão para `int` da função `obtem_valor`) que pede para usuário um valor e se o usuário digitar um valor negativo, imprime uma mensagem e retorna um valor positivo.

```
def obtem_valor_positivo() :
   valor = int( input('Entre um valor: ') )

   if valor >= 0 :
      return valor

   print('Tornando o valor positivo...')

   return -valor
```

Figura 22.1: obtem-positivo.py

Em uma função que não devolve valores, `return` (sem argumentos) pode ser usado para sair de uma função. O exemplo seguinte, pede instruções ao usuário. Se o usuário reponder **não** (*n* ou *N*), a função termina. Do contrário, ele imprime as instruções e depois termina.

```
#coding=latin-1

def instrucoes() :
    ch = input('Você quer instruções? (s/n): ')

    # Termina se resposta for 'n'
    if ch == 'n' || ch == 'N' :
        return

    # Mostra instruções
    print ('As regras do jogo são . . . ')
    #   .
    #   .
    #   .
    return
```

Figura 22.2: instrucoes.py

O `return` ao final do bloco de sentenças que definem o corpo da função é opcional. Se omitido, a função atingirá o final da função e retornará automaticamente. Note que o `return` é opcional somente para funções que **não** devolvem valores.

22.2 Mais sobre Argumentos

A comunicação entre uma função e o chamador pode ser nas duas direções. Argumentos podem ser usados pelo chamador para passar dados para a função. A lista de argumentos é definida pelo cabeçalho da função entre parênteses. Para cada argumento você precisa especificar o nome do argumento. Se houver mais de um argumento, eles são separados por vírgula. Funções que não possuem argumentos tem lista de argumentos vazia. No corpo da função os *argumentos formais* (ou *parâmetros formais*) são tratados como variáveis. Antes da execução da função os valores passados pelo chamador são atribuídos aos argumentos da função.

Considere o seguinte programa com a função `abs()` que calcula o valor absoluto de um número.

```
#coding=latin-1

# Definicao da funcao abs
def abs(x) :
    if x < 0 :
        x = -x

    return x

# Programa Principal
n = int( input('Entre um numero: ') )

print ('Valor absoluto de ', n, ' é ', abs(n) )
```

Figura 22.3: abs.py

A função `abs()` tem um argumento formal do tipo `int`, e seu nome é `x`. Dentro da função, `x` é usado como uma variável `x` normalmente.

Uma vez que `abs()` tem um único argumento, quando ela é chamada, há sempre um valor dentro do parênteses, como em `abs(n)`. O valor de `n` é passado para a função `abs()`, e antes da execução da função, o valor de `n` é atribuído a `x`.

Aqui está um outro exemplo de uma função que converte uma temperatura de Farenheit para Celsius:

```
def fahr_para_cels(f) :
    return 5.0 / 9.0 * (f - 32.0)
```

Figura 22.4: temperatura.py

Como você pode ver, esta função tem somente um argumento do tipo `float`. Um exemplo de chamada desta função poderia ser:

```
fervura = fahr_para_cels(212.0)
```

O resultado da função `fahr_para_cels(212.0)` é atribuído a `fervura`. Portanto, depois da execução desta sentença, o valor de `fervura` (que é do tipo `float`) será `100.0`.

O exemplo seguinte possui mais de um argumento:

```
#coding=latin-1

def area(largura, altura) :
    return largura * altura
```

Figura 22.5: area.py

Esta função possui dois argumentos numéricos. Para chamar uma função com mais de um argumento, os argumentos devem ser separados por vírgula. A ordem em que os argumentos são passados deve ser na mesma em que são definidos. Neste exemplo, o primeiro valor passado será a largura e o segundo a altura. Um exemplo de chamada seria

```
tamanho = area(14.0, 21.5)
```

Depois desta sentença, o valor de `tamanho` será `301.0`.

Quando passar os argumentos, é importante ter certeza de passá-los na ordem correta e que eles são do tipo correto. Se isso não for observado, pode ocorrer erro ou aviso de compilação, ou resultados incorretos podem ser gerados.

Uma última observação: Os argumentos que são passados pelo chamador podem ser expressões em geral e não somente constantes e variávies. Quando a função é chamada durante a execução do programa, estas expressões são avaliadas, e o valor resultante é passado para a função chamada.

22.3 Chamada por valor

Considere novamente a função `quad()` em `triReto.py` (Figura 21.6). Se esta função é chamada no código principal como

```
p = quad(x)
```

somente o valor de `x` é passado para `quad()`. Por exemplo, se a variável tem valor 5, para a função `quad()`, `quad(x)` ou `quad(5)` são o mesmo. De qualquer forma, `quad()` receberá somente o valor `5`. `quad()` não sabe se na chamada da função o `5` era uma constante inteira, o valor de uma variável do tipo `int`, ou alguma expressão como `625/25 - 4 * 5`. Quando `quad()` é chamado, não interessa qual a expressão entre parênteses, ela será avaliada e somente o valor passado para `quad()`.

Esta maneira de passar argumentos é chamada de *chamada por valor*. Argumentos em Python são passados por valor. Portanto, a função chamada não pode alterar o valor da variável passada pelo chamador como argumento, porque ela não sabe em que endereço de memória o valor da variável está armazenado.

22.4 Variáveis locais

Como você provavelmente já reparou em alguns exemplos, é possível definir variáveis dentro de funções, da mesma forma que temos definido variáveis dentro do programa principal. A declaração de variáveis é feita no início da função.

Estas variáveis são *restritas* à função dentro da qual elas são definidas. Só esta função pode "enxergar" suas próprias variáveis. Por exemplo:

```
def obtem_int() :
    x = int( input('Entre um valor: ') )
    print('Obrigado!')

# Programa Principal
obtem_int()

# **** Isto esta' errado  ****
print ('Voce digitou ', x)
```

Figura 22.6: obtem-int.py

O programa principal usou um nome `x`, mas `x` não é definido dentro de seu corpo; ele é uma variável local a `obtem_int()`, não ao programa principal. Este programa gera erro de compilação.

Note que é possível ter duas funções que usam variáveis locais com o mesmo nome. Cada uma delas é restrita a função que a define e não há conflito. Analise o seguinte programa (ele está correto):

```
#coding=latin-1

def obtem_novo_int() :
    x = int( input('Entre um valor: ') )
    print('Obrigado!')

    return x

# Programa Principal
x = obtem_novo_int();

# ****Isto nao esta errado !! ****
print('Você digitou', x)
```

Figura 22.7: obtem-novo-int.py

A função `obtem_novo_int()` usa uma variável local chamada `x` para armazenar o valor digitado e retorna como resultado o valor de `x`. O programa principal usa outra variável local, também chamada de `x` para receber o resultado retornado por `obtem_novo_int()`. Cada função tem sua própria variável `x`.

Aula 23

23.1 Retornando diversos valores

Considere o programa abaixo que pede ao usuário dois inteiros, armazena-os em duas variáveis, troca seus valores, e os imprime.

```
#coding=latin-1

# Programa Principal
print('Entre dois numeros: ')

a = int( input() )
b = int( input() )

print('Voce entrou com', a, "e", b)

# Troca a com b
temp = a;
a = b;
b = temp;

print('Trocados eles são', a, "e", b)
```

Figura 23.1: Exemplo-12-1a.py

Aqui está um exemplo de execução do programa:

```
Entre dois números: 3 5
Você entrou 3 e 5
Trocados, eles são 5 e 3
```

O seguinte trecho do programa executa a troca de valores das variáveis `a` e `b`:

```
temp = a;
a = b;
b = temp;
```

É possível escrever uma função que executa esta operação de troca? Considere a tentativa abaixo de escrever esta função:

```
#coding=latin-1

# function troca(x, y)
#  acao:        troca os valores inteiros x e y
#  entrada:     x e y
#  saida:       valor de x e y trocados na origem da chamada da funÃ§Ã£o
#  algoritmo:   primeiro guarda o primeiro valor em um temporario e
#               troca
#
def troca(x, y) :
    temp = x
    x = y
    y = temp

# Programa Principal
print('Entre dois numeros: ')

a = int( input() )
b = int( input() )

print('Voce entrou com', a, "e", b)

# Troca a com b
troca(a, b);

print('Trocados eles são', a, "e", b)
```

Figura 23.2: Exemplo-12-1b.py

Se você executar este programa, verá que ele **não** funciona:

```
Entre dois números: 3 5
Você entrou 3 e 5
Trocados, eles são 3 e 5
```

Como você já viu nas notas anteriores, em Python os argumentos são passados **por valor**. Uma vez que somente os valores das variáveis são passados, não é possível para a função `troca()` alterar os valores de `a` e `b` porque `troca()` não sabe onde na memória estas variáveis estão armazenadas. Além disso, `troca()` não poderia ser escrito usando a sentença `return` porque podemos retornar APENAS UM valor (não dois) através da sentença `return`.

A solução para o problema acima é fazer a função retornar os novos valores de `a` e `b`

Considere novamente o exemplo da função `troca()`. Quando `a` e `b` são passados como argumentos para `troca()`, na verdade, somente seus valores são passados. A função não podia alterar os valores de `a` e `b` porque ela não conhece os endereços de `a` e `b`.

A definição da função `troca()` deve ser alterada, para :

```
#coding=latin-1

# function troca(x, y)
#  acao:        troca os valores inteiros x e y
#  entrada:     x e y
#  saida:       valor de x e y trocados
#  algoritmo:   primeiro guarda o primeiro valor em um temporario e
#               troca
#
def troca(x, y) :
    temp = x
    x = y
    y = temp

    return x, y


# Programa Principal
print('Entre dois numeros: ')

a = int( input() )
b = int( input() )

print('Voce entrou com', a, "e", b)

# Troca a com b
a,b =  troca(a, b);

print('Trocados eles são', a, "e", b)
```

Figura 23.3: Exemplo-12-1c.py

A saída deste programa é:

```
Entre dois numeros: 3 5
Voce entrou com  3 e 5
Trocados, eles sao 5 e 3
```

23.2 Listas como argumentos de funções

Para passar uma lista como argumento (com todos os seus elementos) de uma função passamos o nome do lista. Considere o exemplo abaixo:

```
#coding=latin-1

# função lista_max(a)
#  ação:      acha o maior inteiro de um lista de TAMANHO elementos
#  entrada:   lista a de inteiros
#  saida:     o maior valor do lista
#  suposições: a tem TAMANHO elementos
#  algoritmo:  inicializa max com o primeiro elemento do lista; em
#              uma repeticao compara o max com todos os elementos
#              do lista em ordem e muda o valor de max quando um
#              elemento do lista for maior que o max ja' encontrado.
#
def lista_max(a):
    # Achar o maior valor do lista
    max = a[0]
    for n in a :
        if max < n :
            max = n

    return max

# Programa principal
valor=[]

num = input("Entre um inteiro: ")
while num :
    valor.append(int(num))
    num = input("Entre um inteiro: ")

print ("O maior é", lista_max(valor))
```

Figura 23.4: lista-max.py

Aqui está um exemplo de execução deste programa

```
Entre um inteiro: 73
Entre um inteiro: 85
Entre um inteiro: 42
Entre um inteiro: -103
Entre um inteiro: 15
Entre um inteiro:
O maior e' 85
```

A chamada para `lista_max()` tem `valor` como seu argumento, que é copiado para o parâmetro formal `a`, que é uma lista de inteiros. Note que o tamanho não foi especificado, somente o nome do lista, `a`.

Até este ponto, parece que não há diferença entre passar uma variável simples e um lista como argumento para uma função. Mas há uma diferença fundamental: **QUANDO DEFINIMOS UMA LISTA COMO ARGUMENTO FORMAL, ALTERAÇÕES NA LISTA FEI-**

TAS DENTRO DA FUNÇÃO ALTERAM O CONTEÚDO DA LISTA PASSADA COMO PARÂMETRO REAL NA CHAMADA DA FUNÇÃO.

Para ilustrar este conceito, considere o exemplo seguinte:

```
#coding=latin-1

# Troca valor do parâmetro
def troca(a) :
    a = 20

# Troca valores de elementos em um vetor
def troca_vet(vet) :
    vet[0] = 60
    vet[1] = 70
    vet[2] = 80

# Programa Principal
x = 10
v = [30, 40, 50]

troca( x )
print ("x =", x)

print ("ANTES DE troca_vet(): ", end=" ")
print ("v[0] =", v[0], " v[1] =", v[1], " v[2] =", v[2])

troca_vet( v )

print ("DEPOSI DE troca_vet(): ", end=" ")
print ("v[0] =", v[0], " v[1] =", v[1], " v[2] =", v[2])
```

Figura 23.5: troca-01.py

A saída deste programa é:

```
x = 10
ANTES DE troca_vet():  v[0] = 30  v[1] = 40  v[2] = 50
DEPOSI DE troca_vet():  v[0] = 60  v[1] = 70  v[2] = 80
```

O valor da variável `x` do programa principal não se altera porque como já vimos na aula 21 , quando a função `troca` é chamada, o valor do argumento real `x` é avaliado, que é `10`, este valor é copiado para o parâmetro formal `a` da função `troca` e a função então é executada. O parâmetro `a` da função é tratada como variável local, portanto quando atribuímos `20` a `a`, estamos atribuindo `20` a uma variável local. Terminada a função, a execução retorna ao programa principal, que imprime o valor de `x`, que não foi alterado, ou seja, imprime `x = 10`.

Quando a função `troca_vet` é chamada, a lista `v` é passada como argumento e "associada" ao parâmetro formal `vet`. A função é então executada, e os elementos do lista são alterados para `60, 70, 80`. Como mencionado anteriormente, quando passamos uma lista como parâmetro, as

alterações feitas na lista dentro da função alteram a lista passada como parâmetro. Portanto, quando a função termina e a execução continua no programa principal com a impressão dos valores dos elementos de `v`, será impresso `60, 70, 80`, os novos valores alterados de dentro da função `troca_vet`.

Observe que se passarmos como argumento apenas um elemento do lista, o comportamento é o mesmo que se passássemos uma variável simples. Ou seja, o nome do lista indexado por um valor entre colchetes refere-se ao valor do elemento do lista, enquanto o nome do lista sozinho refere-se à lista completa. Assim, no programa abaixo:

```
#coding=latin-1

# Troca valor do parâmetro
def troca(a) :
    a = 20

# Troca valores de elementos em um vetor
def troca_vet(vet) :
    vet[0] = 60
    vet[1] = 70
    vet[2] = 80

# Programa Principal
v = [30, 40, 50]

troca( v[0] )
print ("v[0] =", v[0])
troca_vet( v )
print ("v[0] =", v[0], " v[1] =", v[1], " v[2] =", v[2])
```

Figura 23.6: troca-02.py

A saída do programa é:

```
v[0] = 30
v[0] = 60 v[1] = 70 v[2] = 80
```

Outro exemplo: a função `inicializaLista` abaixo inicializa todos os elementos do lista `valor` com um valor passado como argumento pelo programa principal.

CodigoCinicializa-listavetores

Como as alterações feitas por `inicializaLista` são vistas do programa principal, depois da função `inicializaLista` ser executada, no programa principal todos os elementos do lista `valor` terão o valor 42.

23.3 Comentários

Você pode colocar comentários no seu programa para documentar o que está fazendo. O compilador ignora completamente o que quer esteja dentro de um comentário.

Comentários em Python começam com um `#`. Alguns exemplos:

```
# Este é um comentario sem graca

# Este é
# um comentario
# que usa
# diversas linhas
#
```

Regras para comentário

É sempre uma boa idéia colocar comentários em seu programa das coisas que não são claras. Isto vai ajudar quando mais tarde você olhar o programa que escreveu já há algum tempo ou vai ajudar a entender programas escritos por outra pessoa.

Um exemplo de comentário útil:

```
# converte temperatura de farenheit para celsius
celsius = (fahrenheit - 32) * 5.0 / 9.0
```

O comentário deve ser escrito em português e não em Python . No exemplo abaixo

```
# usando input(), obter valor de idade e multiplicar
# por 365 para obter dias
#
idade = int( input() )
dias = idade * 365
```

o comentário é basicamente uma transcrição do código do programa. Em seu lugar, um comentário como

```
# obtem idade e transforma em numero de dias
```

seria mais informativo neste ponto. Ou seja, você deve comentar o código, e não codificar o comentário.

Você também deve evitar comentários inúteis. Por exemplo:

```
# Incrementa i
i = i + 1
```

Não há necessidade de comentários já que `i = i+1` já é auto explicativo.

Documentação de funções

Você deve documentar as funções que escreve. Na documentação você deve especificar as seguintes informações:

Ação – o que a função faz

Entrada – descrição dos argumentos passados para a função

Saída – descrição do valor retornado pela função

Suposições – o que você assume ser verdade para que a função funcione apropriadamente

Algoritmo – como o problema é resolvido (método)

Estas informações devem ser colocadas como comentário antes do cabeçalho de definição da função conforme o exemplo abaixo:

```
  # funcao instrucoes()
  #  acao:        mostra instrucoes do programa
  #  entrada:     nenhuma
  #  saida:       nenhuma
  #  suposicoes:  nenhuma
  #  algoritmo:   imprime as instrucoes
  #
 def instrucoes() :
    # mostra instrucoes
    print('O processo de purificação do  Uranio-235 é . . .  ';
      .
      .
```

23.4 Exercícios de Fixação

1. **(calcMedia)** Faça programa que obtenha do usuário um conjunto de valores reais e calcule a média aritmética dos valores lidos. O usuário indica o final do conjunto de valores digitando um valor qualquer não numérico. A validação da entrada do usuário deve ser feita pela função ehNumerico(), que recebe como parâmetro um *string* e verifica se esta *string* representa um valor numérico (inteiro ou real), retornando True se for o caso, ou False se não for o caso.

2. **(dna)** Calcular a porcentagem dos nucleotídeos G e C em diversas cadeias de DNA fornecidas pelo usuário. O programa termina quando uma cadeia vazia é informada. O programa deve usar a função conta_gc() que recebe como parâmetro uma sequência de DNA e retorna a porcentagem de nucleotídeos G e C nesta sequencia.

3. **(pesoIdeal)** Faça um programa que obtém do usuário um conjunto de pares de valores (sexo, altura) representando o sexo (M ou F) e altura de pessoas. Para cada um, mostre o peso ideal da pessoa, que deve ser calculado pela função pesoIdeal(), a ser definida pelo programador. A função deve calcular e devolver o peso ideal de uma pessoa dados seu sexo e altura de acordo com as fórmulas abaixo:

$$(72,7 \times altura) - 58 \quad (para \;\; homens)$$
$$(62,1 \times altura) - 44,7 \quad (para \;\; mulheres)$$

   O programa termina quando for informado sexo inválido ou altura 0 (zero).

4. **(bolsa)** Deseja-se distribuir bolsas de estudo para alunos de um curso. O valor da bolsa depende do valor da nota que o aluno teve em uma certa disciplina, conforme tabela abaixo:

Nota em disciplina	Bolsa (R$)
$70 \leqslant$ nota < 80	500,00
$80 \leqslant$ nota < 90	600,00
nota $\geqslant 90$	700,00
$70 >$ nota	0,00 (sem bolsa)

   Pede-se um programa que ao receber a nota de 3 alunos, escolha a maior nota e conforme o valor desta, imprima na tela o valor da bolsa correspondente. Defina e use a função maior() para encontrar o maior valor de três números que são passados como parâmetros. Também deve ser definida e usada a função bolsa() para calcular e devolver o valor da bolsa para uma determinada nota que é passada como parâmetro.

23.5 Exercícios

1. **(lancheria)** O cardápio de uma lancheria é o seguinte:

Especificação	Código	Preço
Cachorro quente	100	1,20
Bauru simples	101	1,30
Bauru com ovo	102	1,50
Hambúrger	103	1,20
Cheeseburguer	104	1,30
Refrigerante	105	1,00

   Escrever um programa que leia um conjunto de pares de valores que indicam o código de um item e a quantidade do item e calcule o valor TOTAL. O cálculo do preço de um item deve ser feito pela função calcLanche() que recebe como parâmetros o código de um lanche e sua quantidade e retorna o sub-total de acordo com a tabela acima. Se o código informado é inválido, a função deve imprimir uma mensagem de erro, informando o código incorreto.
   O programa termina quando for informado o código de valor 0 (zero).

2. **(palindromo)** Faça programa que recebe vários conjuntos de 2 inteiros positivos quaisquer do usuário e indica se um é o contrário do outro, isto é, se o primeiro número for re-escrito do último dígito para o primeiro dá como resultado o segundo número. O programa termina quando o primeiro número de um conjunto for negativo. A verificação da propriedade deve ser feita pela função ehPalindromo() que recebe 2 *strings* como parâmetros e retorna True se as 2 *strings* possuem a propriedade e False caso contrário.

3. **(abrevia)** Escreva programa que obtenha do usuário vários nomes. Para cada um deles, o programa deve imprimir na tela todas as partes do nome abreviadas, com excessão do último sobrenome. Além disso, as primeiras letras de todas as partes devem estar em MAIÚSCULA. Caso o nome informado tiver apenas 1 parte, ele deve ser considerado inválido e a conversão não deve acontecer. Quando o usuário digitar a palavra 'FIM', o programa deve imprimir quantos nomes foram informados e quantos nomes não foram processados. A palavra 'FIM' não deve ser processada de forma alguma (conversão e contagem).
   Ex.: se entrada for 'Armando Luiz Nicolini Delgado', a saída deve ser 'A.L.N. Delgado'
   O programa deve usar a função abrevia(), que recebe uma *string* contendo um nome a ser abreviado e retorna o nome abreviado conforme indicado acima, ou retorna a palavra *Invalido* se o nome não puder ser abreviado.

4. **(dna-2)** Calcular a porcentagem de nucleotídeos informados pelo usuário em diversas cadeias de DNA também fornecidas pelo usuário. O programa termina quando uma cadeia vazia é informada. O programa deve usar a função conta_nucl() que recebe como parâmetros uma sequência de DNA e um conjunto de nucleotídeos retorna a porcentagem de cada nucleotídeo na sequencia.

Tópico 10

Arquivos

Aula 24

24.1 Introdução

O armazenamento de dados em variáveis e outras estruturas de dados (listas, dicionários, etc) é temporário. Arquivos são usados para armazenamento permanente de grandes quantidades de dados em dispositivos de armazenamento secundário, como discos.

Às vezes não é suficiente para um programa usar somente a entrada e saída padrão (teclado e tela do computador). Há casos em que um programa deve acessar arquivos. Por exemplo, se nós guardamos uma base de dados com endereços de pessoas em um arquivo, e queremos escrever um programa que permita ao usuário interativamente buscar, imprimir e mudar dados nesta base, este programa deve ser capaz de ler dados do arquivo e também gravar dados no mesmo arquivo.

No restante desta seção discutiremos como arquivos de texto são manipulados em Python . Como será visto, tudo ocorre de maneira análoga ao que acontece com entrada e saída padrão (teclado e tela do computador).

24.2 Preparando arquivo para uso: **open()**

Python visualiza cada arquivo simplesmente como um fluxo (*stream*) seqüencial de bytes. Em Python um arquivo termina termina com um marcador de final de arquivo (*end-of-file*), `EOF`.

As regras para acessar um arquivo são simples. Antes que um arquivo seja lido ou gravado, ele é *aberto*. Um arquivo é aberto em um *modo* que descreve como o arquivo será usado (por exemplo, para leitura, gravação ou ambos). Um arquivo aberto pode ser processado por funções da biblioteca padrão em Python . Estas funções são similares às funções de biblioteca que lêem e escrevem de/para entrada/saída padrão. Quando um arquivo não é mais necessário ele deve ser *fechado*. Ao final da execução de um programa todos os arquivos abertos são automaticamente fechados. Existe um número máximo de arquivos que podem ser simultaneamente abertos de forma que você deve tentar fechar arquivos quando você não precisa mais deles.

Quando um arquivo está *aberto*, um fluxo (*stream*) é associado ao arquivo. Este *stream* fornece um canal de comunicação entre um arquivo e o programa. Três arquivos e seus respectivos *streams* são abertos automaticamente quando um programa inicia sua execução: a

entrada padrão (teclado), a saída padrão (tela do computador) e a saída padrão de erros (tela do computador).

A função padrão open() é usada para abrir um arquivo. open() toma dois argumentos do tipo *string*: o primeiro argumento é o nome do arquivo (por exemplo data.txt), o segundo argumento é a indicação do modo no qual o arquivo deve ser aberto. fopen() negocia com o sistema operacional e retorna um um valor especial que representa o arquivo aberto. Este valor é chamado *file handle*, e representa uma estrutura de dados interna que contém informações de sistema sobre o arquivo. O *file handle* possui um conjunto de funções associadas (análogo ao que acontece ao tipo *string*) que processam o arquivo aberto, e "representa"o arquivo do momento em que é aberto até o momento em que é fechado.

O usuário não necessita saber detalhes de como a transferência de dados entre programa e arquivo é feita. As únicas sentenças necessárias no programa são a definição de uma variável do tipo *file handle* e uma atribuição de um valor para aquela variável por open(). As sentenças abaixo dão um exemplo de como abrir o arquivo data.txt para leitura.

```
infh = open('data.txt', 'r')
```

A especificação da função open() é:

```
open(nome_arquivo, modo)
```

open() recebe dois argumentos: o primeiro é uma *string* representando o nome de um arquivo a ser aberto, e o segundo é uma *string* que representa o modo de abertura do arquivo conforme a tabela

Modo	Descrição
r	Apenas leitura (não consegue gravar dados). Se o arquivo não existe, é gerada excessão do tipo IOError.
w	Apenas escrita (não consegue ler dados). Se o arquivo não existe, ele é criado com conteúdo vazio. Se o arquivo existe, seu conteúdo é DESCARTADO.
a	Grava a partir do final do arquivo (*append*). Se o arquivo não existe, ele é criado com conteúdo vazio.
[rwa]b	Operações de leitura, escrita e acréscimo em modo binário (necessário no s.o. Windows).
r+	Leitura E escrita. Se o arquivo não existe, é gerada excessão do tipo IOError.
w+	Escrita E leitura. Se o arquivo não existe, ele é criado com conteúdo vazio. Se o arquivo existe, seu conteúdo é DESCARTADO.

Tabela 24.1: Modos de Operação em Arquivos

Cada arquivo aberto possui seu próprio *file handle*. Por exemplo, se um programa vai manipular dois arquivos diferentes arq1 e arq2 simultaneamente (um para leitura e outro para escrita), dois *file handles* devem ser usados:

```
fp1 = open('arq1', 'r')
fp2 = open('arq2', 'w')
```

Os valores de um *file handle* estabelecem conexão entre o programa e o arquivo aberto. A partir do momento de abertura, o nome do arquivo é irrelevante para o programa. Todas as funções que operam sobre o arquivo usam o *file handle* associado.

24.3 Terminando as operações sobre arquivo: close()

Terminada a manipulação do arquivo o programa deve fechar o arquivo. A função close() associado a um *file handle* é usada com este propósito. Ela quebra a conexão entre o *file handle* e o arquivo. Abaixo um exemplo de uso de close():

```
fp1.close()
fp2.close()
```

É importante salientar que, se houve operações de escrita em um arquivo, as alterações somente serão efetivadas, isto é, fisicamente gravadas no arquivo após a execução da função close().

24.4 Lendo e escrevendo em arquivos

Arquivos podem guardar duas categorias básicas de dados: **texto** (caracteres no universo ASCII) ou **binário** (como dados armazenados em memória ou dados que representam uma imagem JPEG). Estes últimos devem ter a letra 'b' ao final da *string* que representa o modo de operação na função open() (veja Tabela 24.1)

Depois que um arquivo é aberto, existem diversas formas diferentes de ler ou escrever sequencialmente os dados: (i) um caracter por vez; (ii) uma linha (*string*) por vez; e (iii) em um formato específico.

Arquivos binários podem ser lidos como registros de dados estruturados (vetores, listas, tuplas, etc.). Além disso, se todos os registros tem o mesmo tamanho, os dados podem ser acessados de forma não-sequencial (acesso aleatório).

Entrada e saída de textos (*strings*)

As funções usadas para ler o conteúdo de um arquivo somente podem ser usadas em arquivos que tenham sido abertos nos modos r e correlatos.

As especificações destas funções são:

Função	Descrição
readline([n])	A função readline() retorna uma *string* contendo 1 (uma) linha de texto lida do arquivo conectado ao *stream* `fp`, parando a leitura se o caracter de mudança de linha é encontrado. Este caracter `'\n'` é incluído na leitura. O valor n é opcional e se estiver presente, é feita a leitura de NO MÁXIMO n caracteres da linha. A função retorna uma *string* vazia se um erro ou final de arquivo ocorre.
readlines()	Gera uma lista em que cada elemento é uma linha do arquivo.

Tabela 24.2: Funções para leitura de arquivos

As funções usadas para gravar dados em um arquivo somente podem ser usadas em arquivos que tenham sido abertos nos modos w, a e correlatos.

As especificações destas funções são:

Função	Descrição
write (str)	A função write() grava a *string* str no arquivo conectado ao *stream* fp.
writelines(lst)	A função writelines() grava cada elemento da lista lst no arquivo conectado ao *stream* fp.

Tabela 24.3: Funções para escrita em arquivos

Nos exemplos seguintes são mostradas a forma simples de gravar dados em um arquivo e e 3 formas simples para ler dados do mesmo arquivo.

```
# ************************************************************
# * Escreve texto lido pelo teclado em um arquivo
#*************************************************************


# Programa Principal

# abre arquivo de saida. Conteúdo anterior é perdido.
fp = open('saida.txt', 'w' )

# lê linha do teclado, armazena em uma string,
# salva string em arquivo
linha = input()
while linha :
    fp.write (linha)
    linha = input()

# fecha arquivo
fp.close()
```

Figura 24.1: escrever.py

```
# ************************************************************
# * Ler de um arquivo e exibir na tela
#************************************************************

# Programa Principal

# abre arquivo para leitura.
fp = open('saida.txt', 'r' )

# lê cada linha do arquivo e imprime na tela
linha = fp.readline()
while linha :
    print(linha)
    linha = fp.readline()

# fecha arquivo
fp.close()
```

Figura 24.2: ler-1.py

```
# ************************************************************
# * Ler de um arquivo e exibir na tela
#************************************************************

# Programa Principal

# abre arquivo para leitura.
fp = open('saida.txt', 'r' )

# lê cada linha do arquivo e imprime na tela
for linha in fp.readlines():
    print(linha)

# fecha arquivo
fp.close()
```

Figura 24.3: ler-2.py

```
# ************************************************************
# * Ler de um arquivo e exibir na tela
#************************************************************

# Programa Principal

# abre arquivo para leitura.
fp = open('saida.txt', 'r' )

# lê cada linha do arquivo e imprime na tela
for linha in fp :
    print(linha)

# fecha arquivo
fp.close()
```

Figura 24.4: ler-3.py

24.5 Exercícios de Fixação

1. **(calcmedia-01)** Faça programa que obtenha de um arquivo chamado calcmedia.txt um conjunto de valores reais (um valor em cada linha) e calcule a média aritmética dos valores lidos. O programa termina a leitura quando não houver mais valores a serem lidos do arquivo.

2. **(lancheria-01)** O cardápio de uma lancheria é o seguinte:

Especificação	Código	Preço
Cachorro quente	100	1,20
Bauru simples	101	1,30
Bauru com ovo	102	1,50
Hambúrger	103	1,20
Cheeseburguer	104	1,30
Refrigerante	105	1,00

   Escrever um programa que leia do arquivo lanches.txt um conjunto de pares de valores representando o código de um item e a quantidade do item (um par de valores por linha) e calcule o valor TOTAL. O programa termina a leitura ao terminarem os dados do arquivo de entrada.

Aula 25

25.1 Mais sobre nomes de arquivos

Falar sobre caminho relativo e absoluto. Falar sobre conceito de diretório x pasta (windows). Explicar como funciona em Windows e Linux e MacOS

Entrada e saída formatada

Como as funções de gravação operam com *strings* , usa-se a formatação de *strings* vista na aula 5 , seção 5.3.

```
#! /usr/bin/env python3
# coding=latin-1

from string import *
from math import *

# abre arquivo de saida. Conteúdo anterior é perdido.
fp = open('saida.txt', 'w' )

PRECO, qtde, qualidade, peso = input("Indique preço, qtde., qual., e peso: ").split()
PRECO = float(PRECO)
qtde = int(qtde)
peso = float(peso)

fp.write ('{0} {1} {2:.3f}\n'.format (qtde, qualidade, peso))
fp.write ('{0:.2f} {1:.2f}\n'.format (PRECO, peso * PRECO))

# fecha arquivo
fp.close()
```

Figura 25.1: gravaFormatado.py

A leitura de valores com formatação faz-se tratando-se a *string* lida do arquivo.

```
#! /usr/bin/env python3
# coding=latin-1

from string import *
from math import *

# Programa Principal

# abre arquivo para leitura.
fp = open('saida.txt', 'r' )

# lê cada linha do arquivo e imprime na tela
qtde,qualidade,peso = fp.readline().split()
preco, total = fp.readline().split()

print('Total: R$ {0:.2f}'.format(int(qtde) * float(total)))
print('Peso total: {0:.3f} kg'.format(int(qtde) * float(peso)))

# fecha arquivo
fp.close()
```

Figura 25.2: lerDados.py

25.2 Exercícios

1. **(abreviacoes)** Escreva programa que obtenha de um arquivo vários nomes. Para cada um deles, o programa deve gravar em outro arquivo o último sobrenome com todas as letras maiúsculas, seguido de vírgula, seguido do restante do nome, onde apenas o 1o. nome deve estar por extenso e o restante abreviado. Caso o nome informado tiver menos que 3 partes (1 nome e 2 sobrenomes, por exemplo), ele não deve ser processado, isto é, não deve ocorrer conversão.
   Quando não houver mais nomes no arquivo, o programa deve imprimir na tela quantos nomes foram informados e quantos nomes não foram processados.
   Exemplo de execução:

```
Conteúdo do arquivo de saída (não aparece na tela)
  NOME: Armando Luiz Nicolini Delgado

    ABNT: DELGADO, Armando L. N.

  NOME: Roberto Almeida Prado

    ABNT: PRADO, Roberto A.

  NOME: Aluisio Feirante

    Não Processado

  NOME: Delagado

    Não Processado


Mensagem na tela
  Foram indicados 4 nomes, dos quais 2 não foram processados.
```

2. **(achaPadroes)** Faça um programa que obtenha de um arquivo um conjunto de sequencias de nucleotídeos (uma sequencia por linha). Para cada sequencia informada, o programa deve ler de outro arquivo um conjunto de padrões a serem encontradas em cada sequencia. Quando não houver mais padrões, o programa busca no primeiro arquivo nova sequencia para análise. Quando este não tiver mais sequencias, o programa termina. Para cada padrão, o programa deve listar TODAS as posições em que foi encontrado dentro da sequencia em análise. Caso o padrão não ocorra na sequencia, o programa deve indicar tal fato com uma mensagem adequada.

3. **(conta)** Escreva um programa que obtenha de um arquivo chamado genbank.dat que contém vários conjuntos de 2 linhas onde: a 1a. linha do conjunto contém 2 (dois) valores inteiros separados por espaço, e a 2a. linha contém uma sequência de nucleotídeos.

   Para cada conjunto lido do arquivo, o programa deve imprimir na tela o trecho da sequência contido no intervalo indicado pelos dois valores inteiros e em seguida indicar a proporção

de cada nucleotídeo (A,C,T,G) naquele trecho. Para efeitos deste exercício, o arquivo considera o valor 1 como sendo o primeiro caracter da sequencia.

A proporção de cada nucleotideo deve ser calculada pela função proporcao() que recebe dois parâmetros: uma sequência `seq` qualquer de nucleotídeos e um nucleotideo `nucleot`, retornando a proporção de ocorrências do nucleotídeo dentro da sequência. A função deve retornar o valor 0 (zero) caso o nucleotídeo não for encontrado.

Exemplo de conteúdo do arquivo:

```
20 47
gatcctccatatacaacggtatctccacctcaggtttagatctcaacaacggaaccattgccgacatgagacagtt
10 70
ctactatatcactactccatctagtagtggccacgccctatgaggcatatcctatcggaaaacaataccccccagt
5 35
tccagcgtggatgaaaagagagattctctatcaggtatgaatacatacaatgatcagttccaatcccaaagtaaag
```

Exemplo de saída do programa:

```
tatctccacctcaggtttagatctcaac
A=25.00%, C=32.14%, G=10.71%, T=32.14%
cactactccatctagtagtggccacgccctatgaggcatatcctatcggaaaacaataccc
A=29.51%, C=32.79%, G=16.39%, T=21.31%
gcgtggatgaaaagagagattctctatcagg
A=32.26%, C=12.90%, G=32.26%, T=22.58%
```

Tópico 11

Tratamento de Excessões

Aula 26

26.1 Excessões

Em Python , quando ocorre um erro durante a execução do programa que a definição da linguagem considera como um erro sério, o programa cessa sua execução e uma mensagem detalhada da causa da interrupção é exibida na tela. Estas causas incluem divisão por zero, uso de valores inadequados para índices de coleções, e falha na manipulação de arquivos.

Em Python , estes erros são chamados de **excessões** e é possível tratar as ocorrências destas causas dentro do programa, de forma que o próprio programador decide o que fazer, ao invés de ter o programa interrompido.

Por exemplo, seja o programa `erroLista-00.py` (Figura 26.1):

```
seq = [1, 2, 3, 4, 5, 6]

i = 6

val = seq[i] * 7

print(val)
```

Figura 26.1: `erroLista-00.py`

Ao ser executado, a seguinte mensagem é mostrada na tela:

```
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
IndexError: list index out of range
```

Durante sua execução, a primeira atribuição gera uma excessão do tipo **IndexError**, pois o índice usado (6) é maior do que o índice do último elemento da lista *seq* (5).

Como outro exemplo, seja o programa `erroLista-01.py` (Figura 26.2):

```
seq = [1, 2, 3, 4, 5, 6]

val = 10

idx = seq.index(val)

print(idx)
```

Figura 26.2: erroLista-01.py

Ao ser executado, a seguinte mensagem é mostrada na tela:

```
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
ValueError: list.index(x): x not in list
```

Durante sua execução, o valor 10 não existe na lista e a execução do método index() gera uma excessão do tipo **ValueError**.

Estas duas excessões podem ser tratadas pelo código do programa através da estrutura de controle try ... except, como podemos ver em erroLista.py (Figura 26.3):

```
#coding=latin-1

seq = [1, 2, 3, 4, 5, 6]

i = 6

try :
    a = seq[i] * 7
except IndexError :
    print('Índice {0} inválido.'.format(i))
else :
    val = 10
    try :
        idx = seq.index(val)
    except ValueError :
        print('Não achou valor {0}.'.format(val))
    else :
        print('Deu tudo certo. Programa prossegue a partir deste ponto.')

        # Restante do programa abaixo. Observe a indentação, que deve estar sob
        # o 'else'
        # .
        # .
        # .
```

Figura 26.3: erroLista.py

A estrutura try ... except deve ter PELO MENOS uma parte except, podendo ter mais conforme os tipos de excessões que possam ocorrer dentro da parte do try. A parte else é executada APENAS SE NÃO HOUVE A OCORRÊNCIA DE EXCESSÕES.

Exemplos de fixação

Se a tentativa de abertura resulta em erro, open() gera uma excessão do tipo **IOError**. Algumas das causas possíveis desta excessão são: abrir um arquivo que não existe para leitura, abrir um arquivo indicando um diretório inexistente, abrir um arquivo para escrita quando não há mais espaço disponível em disco, ou abrir um arquivo em diretório onde há restrições nas permissões de acesso.

É recomendável que você verifique em seu programa a ocorrência desta excessão para verificar se houve erro de abertura. O trecho de programa abaixo ilustra como fazê-lo:

```
def abre (file_nome, file_mode) :
    sucesso=True
    fp=0

    try :
        fp = open( file_nome, file_modo )
    except IOError :
        sucesso=False

    return sucesso,fp

# Programa principal

stat,inf = abre('teste.jpg', 'r')

if not stat :
    print('Erro  na  abertura de  {0}  no  modo {1}'.format('teste.jpg', 'r'))
else :
    # Arquivo aberto com sucesso no modo leitura

    linha = inf.readline()

    # Restante do programa abaixo. Observe a indentação, que deve estar sob
    # o 'else'
    # .
    # .
    # .
```

Figura 26.4: abreArquivos.py

No exemplo acima a função abre() retorna 2 valores: o primeiro valor indica se houve sucesso ou não na operação open(). O segundo valor somente guarda um *file handler* do arquivo caso haja sucesso na operação. No programa principal, é feita a chamada para a função abre() e é

feito o teste de sucesso e, se for este o caso, o programa lê uma linha do arquivo e o programa continua sua execução.

Outro exemplo se encontra na programa a seguir. A função nFind() procura repetir para listas o comportamento do método find() que existe para *strings*.

```
def nFind (lista, valor) :
    try :
        idx = lista.index(valor)
    except ValueError :
        return -1;

    return idx

# Programa principal

ll = [1,2,3,4,5,6]
ind = nFind(ll, 10)

if ind == -1 :
    print('Não encontrou valor 10')
else :
    print('Encontrou valor 10 no indice {0}.'.format(ind))

    # Restante do programa abaixo. Observe a indentação, que deve estar sob
    # o 'else'
    # .
    # .
    # .
```

Figura 26.5: findLista.py

Aqui, a função nFind() recebe uma lista e um valor a ser encontrado nesta lista, retornando o valor -1 se o valor não for encontrado, ou o índice da primeira ocorrência do valor dentro da lista caso contrário.

Tópico 12

Avaliação 4

Aula 27

27.1 Tema da Avaliação 4

Toda a matéria da disciplina, envolvendo leitura e gravação de dados em arquivos. Os dados a serem lidos são geralmente em formato texto com informações distribuídas em campos (colunas) ou seções de dados.

Esta avaliação pode ser feita em forma de prova individual ou de trabalho em grupos de 2 alunos.

Bibliografia

[1] N.N.C. Menezes. *Introdução à Programação Python*. 2ª edição. Editora Novatec, 2010.

[2] Mark Lutz e D. Ascher. *Aprendendo Python*. 2ª edição. Editora Novatec, 2007.

[3] Magnus Lie Hetland. *Beginning Python: From Novice to Professional*. Springer-Verlag, 2005.

[4] Katja Schuerer, Corinne Maufrais et al. *Introduction to Programming using Python*. Último acesso em 08/08/2011. Pasteur Institute. 2008. URL: http://www.pasteur.fr/formation/infobio/python/.

[5] Marco Medina e Cristina Fertig. *Algoritmos e Programação: Teoria e Prática*. 2a. edição. Novatec Editora Ltda., 2006.