ALGORITMOS EM PYTHON

PRÁTICA DE LABORATÓRIO EM PYTHON

PROF. ELOI LUIZ FAVERO

BELEM - PA
2020

SUMÁRIO

Prefácio

Este texto surgiu como um material de apoio a um curso de tele-ensino de uma disciplina de Algoritmos de 60 a 68 horas. O texto foi elaborado para contemplar a metodologia de ensino descrita a seguir.

Cada capítulo compreende um módulo, tema, de 4 horas aula. São 15 capítulos principais e 3 opcionais. A metodologia é mais assíncrona, porém com encontros síncronos. O essencial de uma disciplina de algoritmos é a prática do estudante, que deve ser verificada, checada a cada módulo, com a correção dos exercícios propostos.

Nos encontros síncronos, os temas e conceitos são apresentados em aulas de 45 min de exposição e um tempo posterior de discussão para tirar dúvidas, respondendo perguntas.

Nas atividades assíncronas, o estudante lê o material dos módulos e resolve os exercícios. No ambiente virtual é feita a postagem de cada módulo. Cada módulo é associado com uma atividade de resolução de exercícios práticos e/ou teóricos. Neste ambiente, o estudante posta a sua resolução dos exercícios e o professor avaliando-os tem feedback para tirar dúvidas e/ou redirecionar os exercícios do próximo módulo.

Os comandos condicionais, de repetição, etc. são inicialmente apresentados através de fluxogramas e pseudocódigo. A seguir, exemplos de usos dos comandos são traduzidos para a linguagem Python e são animados no ambiente Python Tutor. Após o estudante compreender o comportamento do comando apresentado, ele é convidado a executá-los no ambiente de programação da linguagem Python, IDLE.

Segue a lista dos módulos:

1 INTRODUÇÃO & PYTHON TUTOR
2 TIPOS DE DADOS INT E FLOAT
3 TIPO DE DADOS STRING
4 COMANDOS DE ENTRADA E SAÍDA
6 COMANDOS DE DECISÃO (IF)
7 RE-FATORANDO O CÓDIGO
8 LISTA E FAIXA DE VALORES
9 COMANDO DE REPETIÇÂO: FOR
10 COMANDO DE REPETIÇÃO: WHILE
11 FUNÇÕES
12 MÉTODOS DE BUSCA E ORDENAÇÃO
13 VETORES E MATRIZES
14 DICIONÁRIOS E CONJUNTOS: SET E DICT
15 PERFORMANCE DE FUNÇÕES

Este texto destina-se a ensinar algoritmos e programação do ponto de vista de prático. Informação não é conhecimento. Se compro um livro de medicina sobre cirurgia e leio todo o livro, eu tenho o conhecimento para fazer cirurgias? Não. Mas tenho a informação. O que precisa para se transformar informação em conhecimento? Precisa-se da experiência prática. Através da prática de laboratório o aluno adquire as habilidades e competências de programação. Este texto é centrado em exemplos e exercícios para o estudante praticar algoritmos, fixando os conceitos do pensamento computacional. O estudante é convidado a animar os exemplos e exercícios no Python Tutor para depois executá-los numa IDLE como programas.

No total temos mais de 10 exercícios por módulo, mais de 150 exercícios no total. A maior parte dos exercícios é prática de laboratório. Mas, também temos exercícios conceituais para o estudante fixar, formar, estabelecer os conceitos dos temas abordados. Por exemplo, O que é uma variável? O que é um algoritmo? O que é uma função? O que significa refatorar código? É importante o estudante firmar um conjunto de conceitos conforme ele vai amadurecendo no desenvolvimento de algoritmos e programas. Firmar um conceito é amadurecer o suficiente para poder falar sobre o conceito.

O texto aborda algoritmo segundo uma visão de pensamento computacional, onde o pensamento sobre a solução de um determinado problema é organizado numa forma de algoritmo, receita de bolo, que permite a sua resolução dentro de uma linguagem de programação. Um **algoritmo** é uma sequência de comandos que executa uma determinada função computacional. Um **comando** é um passo mínimo dentro de um algoritmo. Comandos são "**verbos**" computacionais que agem sobre variáveis e objetos - fazendo uma analogia com uma sentença de linguagem natural, onde verbos agem sobre sujeitos e objetos. Um algoritmo codificado numa linguagem especifica é um **programa**. Um programa precisa ser **executado** (run) para gerar uma ação.
O livro segue uma abordagem de baixo para cima, do pequeno para o grande, mas também do geral para o especifico. Após a apresentação dos conceitos, vamos nos aprofundar e familiarizar com eles aos poucos. Muitos assuntos são apenas introduzidos para mais adiante serem mais detalhados. Seguindo a abordagem do geral para o mais específico, a cada capítulo novos conceitos são introduzidos e ou antigos conceitos são mais detalhados.
Definimos um kit de ferramentas de pensamento computacional, de pensamento de algoritmos. A tabela abaixo apresenta um kit mínimo, começando por variáveis, comandos de atribuição, comandos de sequência, comandos de entrada e saída, comandos condicionais, comandos de repetição, comando de abstração. Vamos nos aprofundar bem nos comandos de repetição, mostrando várias alternativas para codificar o mesmo algoritmo. Vamos trabalhar com as funções, como uma ferramenta para organizar os códigos através do conceito de abstração, dar nome para um grupo de comandos.

Estruturados	Tipos de comandos	Exemplos
Verticais e/ou horizontais	1-Variáveis, objetos e expressões	a=1; c=[1,2,5,9]
	2-Comando de atribuição	
	3-Sequência de comandos	a=1;b=3+a;x=1+b;print(x)
	4-Comandos de entrada e saida	input(); print()
Só pela indentação	5-Comando condicional	if, elif, else:
	6-Comandos de repetição	for, while
	7-Conceitos de abstração	def

Vamos bater na tecla da refatoração de código, como uma forma de exercitar o raciocínio do pensamento computacional. Fazer diferentes escritas para uma mesma solução de um problema. Muitas vezes quando um programa funcionou pela primeria vez o estudante abandona o seu desenvolvimento. Esta forma de trabalho não é uma boa prática. Uma boa prática é quando o programa funciona reescreve-lo buscando uma versão alternativa (com for, com while, com um dicionário) ou buscando um código mais simples, mais limpo, mais compreensivo, que ainda dê o resultado correto. Esta prática de programação dará a certificação para o estudante, ele tende a perfeição. O que faz, faz bem feito.
Bom estudo ao leitor.

Eloi Favero, Belém 2020

Capítulo

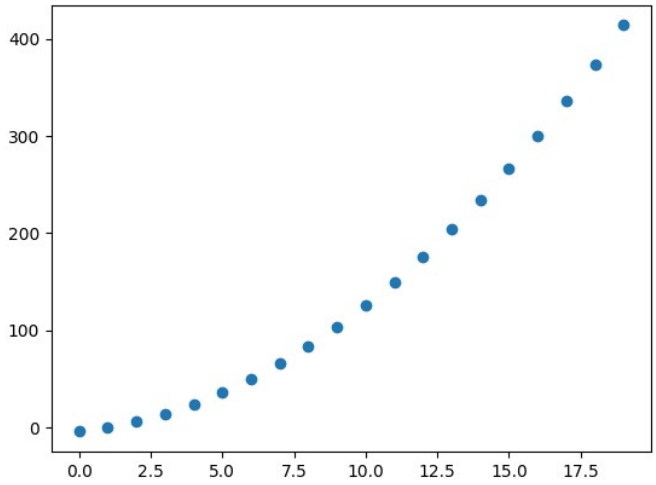
1 INTRODUÇÃO & PYTHON TUTOR

Este capítulo apresenta uma introdução ao conceito de variáveil e ao ambiente Python Tutor para execução de Algoritmos codificados na linguagem Python. Este ambiente mostra a visualização do comportamento do algoritmo, permitindo caminhar passo a passo na execução para frente e para trás sobre as linhas do código. Conceitos apresentados: variável, comando de atribuição, expressões aritméticas, algoritmo, passo de execução, memória, nomes de variáveis. Prática de laboratório com Lista de exercícios.

1.1 Conceito de variável

Estamos familiarizados com o conceito matemático de variável. Por exemplo, seja a equação y=x.x+3x-4. Se x=0 então y=-4. Se x=10 então y=100+30-4=126. E assim por diante. Logo o valor da variável y está definido em função do valor da variável x.

O gráfico abaixo mostra uma lista de valores plotados a partir de um programa Python.




```
from matplotlib import pyplot as plt

X = list(range(0,20))
Y = [x*x+3*x-4 for x in X]

plt.scatter(X,Y)
plt.show()

print(X)
print(Y)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,..., 18, 19]
[-4, 0, 6, 14, 24, 36, 50, 66, 84, 104, 126,...
                                    , 374, 414]
```

Aqui, neste simples programa temos inúmeros conceitos de programação, de pensamento computacional. O range(0,20) é um intervalo fechado no 0 e aberto no 20. X maiúsculo é um vetor (list) de x minúsculos, são os 20 x, [0,1,2,...19]. A equação x.x+3x-4 é representada como x*x+3*x-4. O vetor Y maiúsculo é a coleção de valores resultantes da função aplicada aos 20 valores do vetor X, o primeiro é -4 e o último é 414.

Com este exemplo se entende porque se chamam variáveis, porque os valores variam. Neste exemplo, as variáveis nomeiam valores inteiros ou vetores de valores inteiros. Variáveis em geral armazenam objetos computacionais: valores inteiros, reais, vetores de

inteiros, vetores de reais, nomes, listas de compras, atributos de um ser humano como cpf, data nascimento, altura e cor dos olhos.

Observação: o programa acima não precisa ser compreendido agora. Foi apresentado só para motivar a introdução do conceito de variável. O que é necessário compreender neste módulo é como práticar os exercícios apresentados no final do módulo, sobre o conceito de variável.

1.2 Variáveis em Python

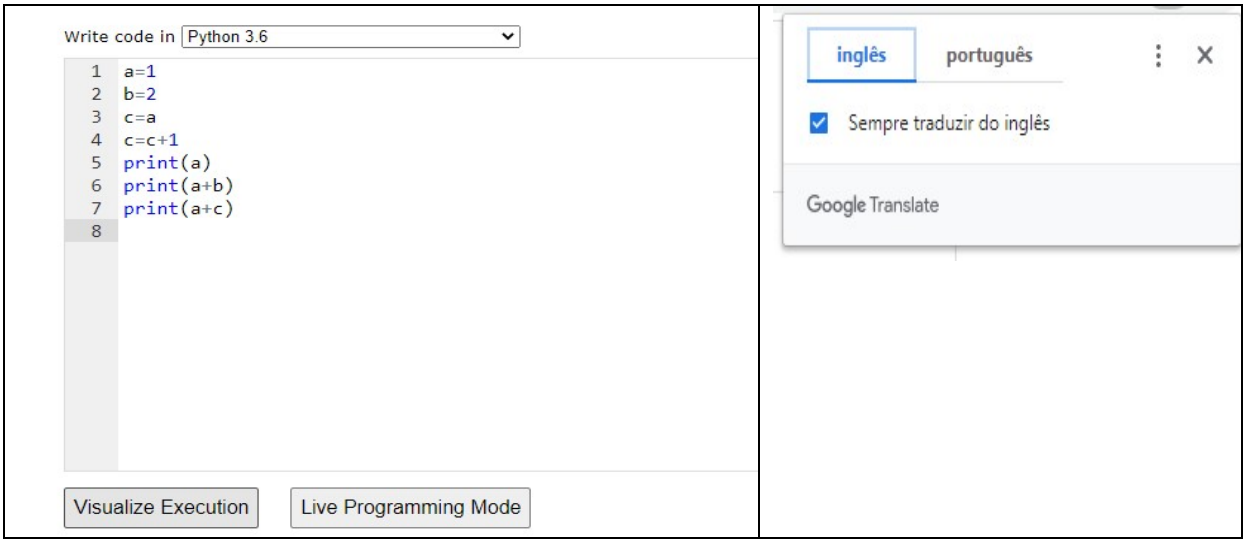
As variáveis ocupam o espaço de memória de um computador. Uma variável em Python é representada por um frame ou por um objeto. Para compreender melhor o que é um frame ou um objeto vamos visualizar a execução de alguns comandos com o Python Tutor, ver na web.
Para quem está aprendendo algoritmos ou iniciando em programação recomendamos o uso do ambiente Python Tutor, pois ele permite a visualização do efeito dos comandos mostrando a representação das variáveis e permitindo o acompanhamento da execução passo a passo dos comandos, para frente e para traz. Visualizando assim o efeito dos comandos sobre as variáveis.
Vamos examinar nosso primeiro programa. Neste exemplo, o símbolo de igual = representa o comando de atribuição. Se a=1 e c=a, então c=1. Depois, c=c+1 então c=1+1. O print() imprime o valor. Só nos próximos capítulos vamos diferenciar o comando de atribuição do teste de igualdade, que é necessário nas expressões de condições.

```
a=1
b=2
c=a
c=c+1
print(a)
print(a+b)
print(a+c)
```

linhas de código

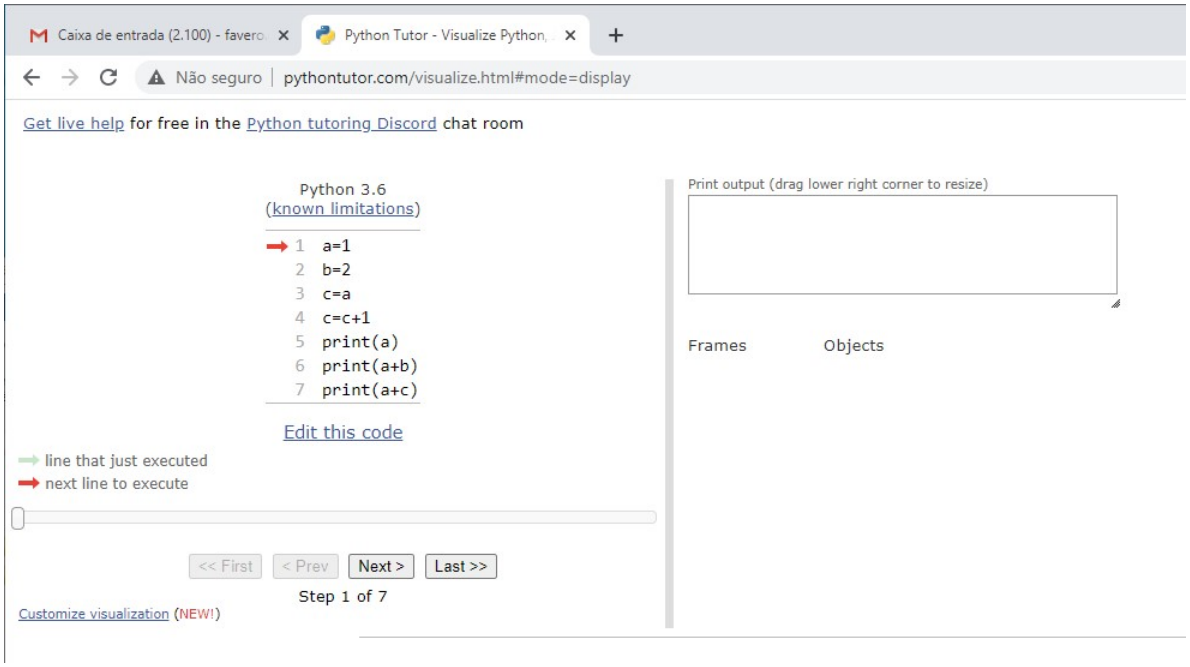
1.3 Python Tutor

Vamos trabalhar com o ambiente online do Python tutor. Marque, copie e cole as linhas de código para uma janela do python tutor. Mantenha o tradutor do google para a língua inglesa. Pois, senão ele vai confundir, vai traduzir só pela metade, e até tentar traduzir palavras do programa.

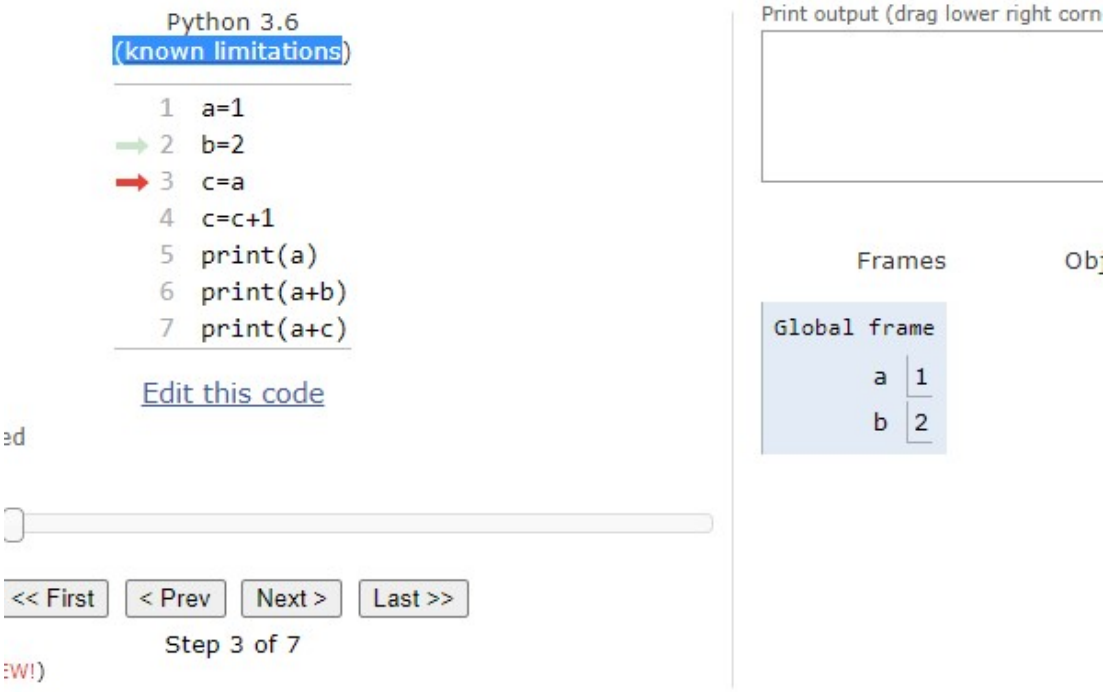


Agora clicando em Visualize Execution, vamos visualizar a execução do código (code). Segue uma tradução para os termos em Inglês, falando dos principais termos usados neste ambiente:

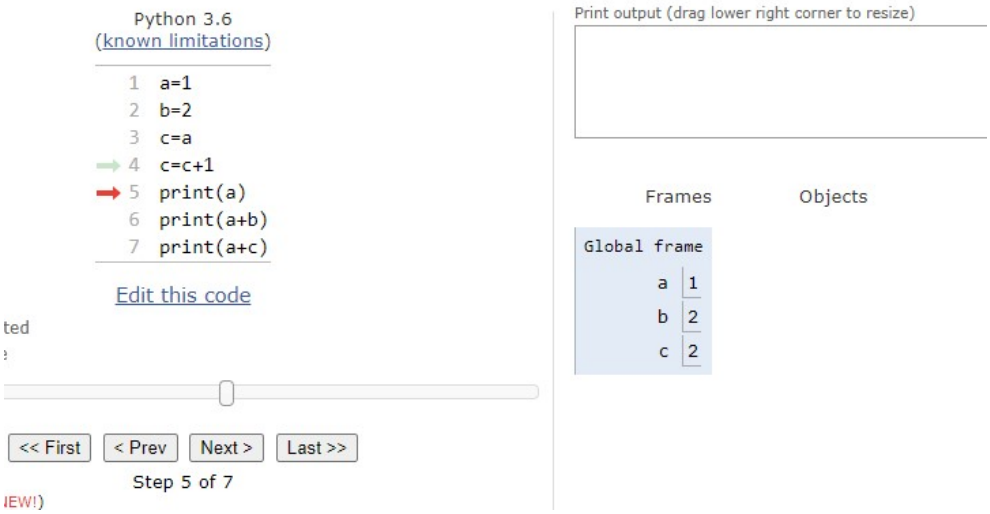
- *Print output (drag lower right corner to resize)* :Escreva saída (araste canto direito inferior para mudar o tamanho)
- *Frames, Objects*: Memorias, Objetos
- *(known limitations)* : limitações conhecidas
- *line that just executed*: linha recém executada
- *next line to execute*: próxima linha a executar
- *Edit this code*: edite este código
- *First, Prev, Next, Last*: primeiro, prévio, próximo, último
- *Global Frame*: Memoria Global



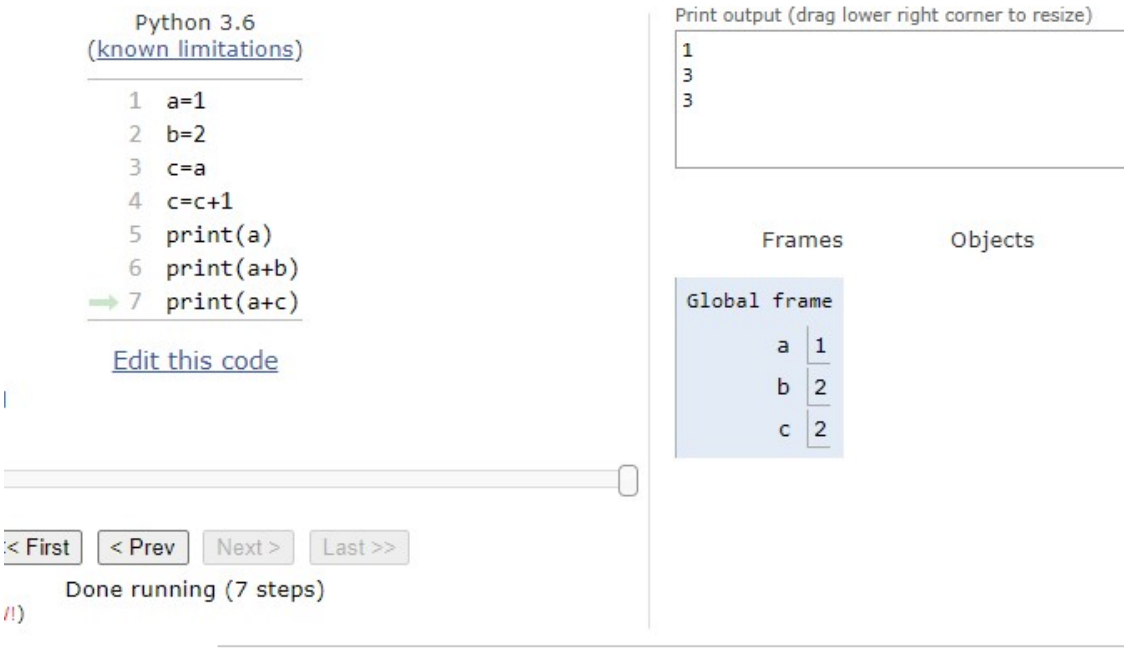
Agora com dois next chegamos na situação abaixo, onde na memória global é mostrado que as variáveis possuem os valores a=1 e b=2.



Com mais um next c=1, e com mais um next c=2 no Global Frame. Podemos executar também para trás com o botão prev. Assim vamos executanto para frete ou para trás o código, até entendermos o que o programa esta fazendo, compreendendo o comportamento da execução do algoritmo.



Nos passos 5, 6,e 7 ocorrem as impressões dos valores na tela de saída, respectivamente os valores 1, 3, 3.



O nome de uma variável sempre começa por uma letra ou pelo _. Depois do primeiro caractere podemos ter também números. A linguagem é **case sensitive**, isto é, nomes com maiúscula e com minúsculas são diferentes, x ≠ X. Os comentários de linha em **Python** são iniciados pelo símbolo #, e vão até o final da linha. Seguem alguns exemplos de declarações de variáveis em comandos de atribuição.

```
# exemplos de variáveis
A1 = 76
a1 = 12 # comentario
aa_1 = a1+A1
zero = 0
```

1.4 Identificação de erros no Python Tutor

Identificação de erros de sintaxe ou de indentação. Abaixo, mostramos um caso de erro de indentação.

Write code in Python 3.6

```
1  # exemplos de variáveis
2  A1 = 76
3   a1 = 12
4  aa_1 = a1+A1
5  zero = 0
6
```

IndentationError: unexpected indent (<string>, line 3)

O tradutor do google pode nos auxiliar na identificação do erro, mas é bom depois voltar para o idioma inglês.

Escreva o código em Python 3.6

```
1  # exemplos de variáveis
2  A1  =  76
3   a1  =  12
4  aa_1  =  a1 + A1
5  zero  =  0
6
```

IndentationError: indentação inesperada (<string>, linha 3)

1.5 Lista de exercícios no python tutor (10)

Vamos adotar a seguinte convenção. Nos exercícios, os códigos dos algoritmos são separados da sua saída pelo prompt >>>, então quando for copiar e colar; copie somente a parte acima do prompt >>>. No próximo capítulo ficará mais claro porque utilizamos o prompt >>>.

Pratique cada um dos exercícios abaixo no python tutor. Trabalhe até compreender o que esta acontecendo com o código, até compreender o comportamento do algoritmo.

E1.1	`a=1`
	`b=2`
	`c=a`
	`c=c+1`
	`print(a)`
	`print(a+b)`
	`print(a+c)`
	`>>>`
	`1`
	`3`
	`3`

E1.2	`# exemplos de variáveis & comentários`
	`A1 = 76`
	`a1 = 12`
	`aa_1 = a1+A1  # o valor de aa_1 é 88`
	`zero = 0`

E1.3	Escreva o resultado de 3a + 5b. Onde a=3 e b=5.
E1.4	Escreva o resultado de 2a x 5b. Onde a=7 e b=11.
E1.5	Escreva o resultado de 4a x -2b. Onde a=3 e b=5.

E1.6	Calcule a soma de três variáveis, a=3;b=5;c=11.
	
	`a=3`
	`b=5`
	`c=11`
	`abc=a+b+c`
	`print(abc)`
	`>>>`
	`19`

E1.7	Seja um salário (sal) de 1000. Como dar um aumento de 15%. Qual o valor do aumento e do novo salário.
	
	`sal=1000`
	`percent=15`
	`aumento=sal*percent/100`
	`novosal=sal+aumento`
	`print(aumento, novosal)`
	`>>>`
	`150.0 1150.0`

E1.8	Seja um salário (sal) de 1500. Como dar um aumento de 20%. Qual o valor do aumento e do novo salário.

	sal=1500 percent=20 aumento=sal*percent/100 novosal=sal+aumento print(aumento, novosal) >>> 300 1800
E1.9	Seja um salário (sal) de 1500. Como dar um aumento de 10%. Qual o valor do aumento e do novo salário.
E1.10	Seja um salário (sal) de 1800. Como dar um aumento de 1%. Qual o valor do aumento e do novo salário.

RESUMO
= comando de atribuição
+ - * / operadores aritméticos
print() comando de impressão
variável
nomes de variáveis
algoritmo é uma sequênciade comandos

Capítulo

2 TIPOS DE DADOS INT E FLOAT

Este capítulo aprofunda o conceito de variável, agora associado as tipos int e float. Apresenta-se o ambiente de programação da linguagem Python. Conceitos de pensamento computacional trabalhados: algoritmo, comandos, tipos de dados, variável, expressões aritméticas, função, programa, arquivo de código. Prática de laboratório com lista de exercícios práticos e conceituais. Conteúdo opcional bit e byte.

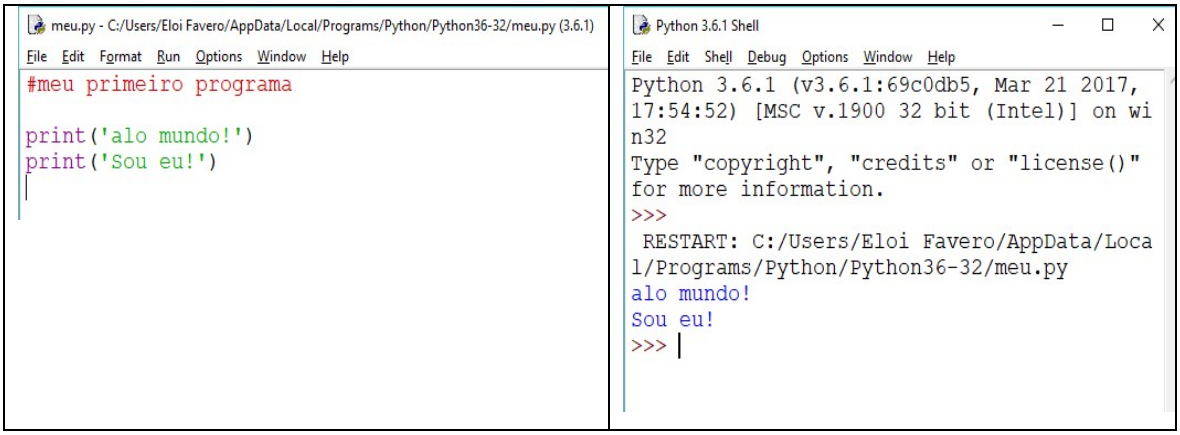
2.1 Alguns conceitos do pensamento computacional

```
sal     = 1500
percent = 20
aumento = sal*percent/100
novosal = sal+aumento
print(aumento, novosal)
```

Um **algoritmo** é uma sequência de comandos que executa uma determinada função computacional. Um **comando** é um passo mínimo dentro de um algoritmo. No exemplo acima, cada linha é um comando. Comandos são "**verbos**" computacionais que agem sobre variáveis e objetos - fazendo uma analogia com uma sentença de linguagem natural, onde verbos agem sobre sujeitos e objetos.
Um algoritmo codificado numa linguagem especifica é um **programa**. No ambiente de programação, um programa em Python normalmente é salvo em um **arquivo** com a **extensão** .py (e.g. meu.py). Um programa precisa ser **executado** (run) para gerar uma ação.

2.2 Ambiente de programação em Python

A linguagem Python tras alguma liberdade ao programador, pois ela pode ser programada num ambiente como o IDLE, PyCharm, etc. como também pode ser programada num ambiente web, por exemplo, o Python Tutor que é indicado para iniciantes.
Na web podemos também programar com um Python notebook, como Jupyter, que permite criar um ambiente web, com texto, código fonte e dados, ambiente indicado para estudantes já iniciados.



Interface do IDLE

Acima, no lado esquerdo criamos o programa **meu.py**. Depois de digitar estas linhas de código o ambiente pede para salvarmos (save) o arquivo antes de executarmos (run) o código. O resultado da execução é mostrado na tela **Shell** do IDLE, no lado direito. No topo da tela, encima, no lado esquerdo, aparece o caminho (path) da pasta onde está o arquivo do programa. Na tela da direita novamente o caminho é repetido dizendo RESTART (re-iniciado, ou re-executado). No lado esquerdo da IDLE temos várias opções de menu: file, edit, format, run, etc.

- No file é importante saber: new (novo arquivo), open (abrir), recent files (abrir um recente), save (salva), save as (salvar como, com novo nome).
- No edit é importante saber: undo (ctr-z, desfazer um erro de edição), replace (substituir um nome de variável por outro); além disso usa-se muito copiar e colar.
- No format é importante saber: indent (fazer uma indentação) e dedent (tirar a indentação). Quando o ambiente diz que tem problema de indentação, uma forma de corrigir é remover toda indentação e criar uma indentação nova e correta, com estes dois comandos.
- No Run é importante saber: run module (F5) rodar o código.

Na tela da direta no shell do IDLE podemos executar diretamente comandos no prompt >>>2*3. Neste lado deve-se executar um comando por vez. Este promt é muito útil para testar operadores e/ou linhas de comandos individuais.

Para usuários iniciantes recomendamos o Python Tutor e o ambiente IDLE. O primeiro é para o estudante compreender o que acontece com as variáveis e objetos usados no algoritmo. O segundo ambiente também é recomendado pela simplicidade. Outros ambientes exigem alguma experiência inicial já praticada nestes ambientes mais fáceis.

Praticando o código no lado direito (shell):

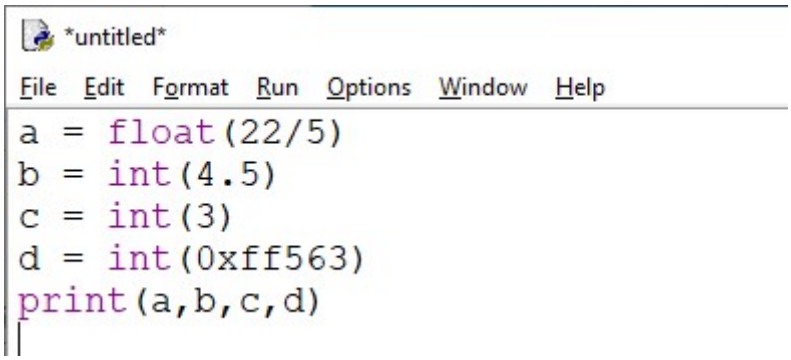
```
# Expressões + - * / % **
>>> 2**3
8
>>> 10 / 3
3.3333333333333335
>>> 10 % 3
1
>>> 8 / 3
2.6666666666666665
>>> 8 % 3
2
```

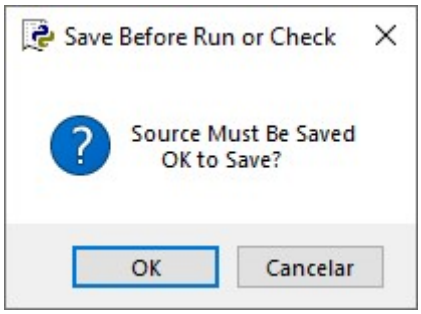
2.3 Tipos de Dados

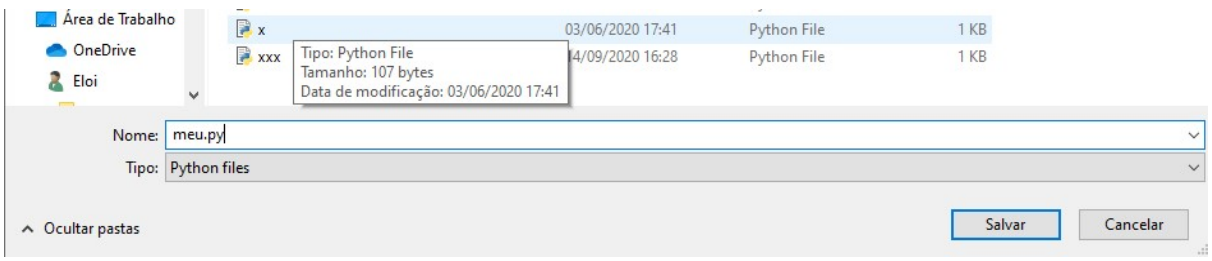
Python tem dois principais tipos de dados numéricos: int e float. Int representa valores inteiros positivos ou negativos. Float representa valores chamados de ponto flutuante ou decimais ou reais. As funções int(), float() e round(), respectivamente inteiro, flutuante e arredondar, permitem fazer a conversão de um tipo de valor para outro tipo de valor.

```
a = float(22/5)
b = int(4.5)
c = int(3)
d = int(0xff563)
print(a,b,c,d)
>>>
4.4 4 3 1045859
```

Esta notação acima, com várias linhas de comandos deve ser digitada no lado esquerdo; em *new file* cria-se uma nova janela *untitled*; depois copia-se e cola-se, não incluir o prompt >>>.
Quando pedimos para rodar (run) o ambiente pede para darmos um nome para o programa ser salvo.





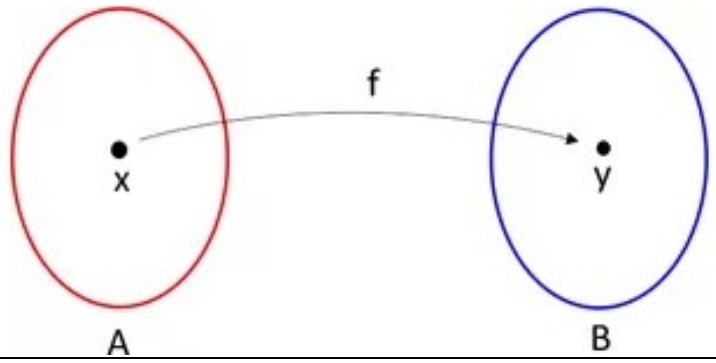


Quando é forçado o *run*, vemos o efeito do código que é o print mostrado depois do prompt. A variável a representa um valor decimal 22/5 = 4.4. A variável b recebe a parte inteira do valor 4.5 que é 4. Em d o valor hexadecimal FF563 equivale ao valor inteiro 1045859.

```
e = float(int(3.9))
f = int(float(3.9))
g = int(float(3))
h = round(3.9)
i = round(3.49)
j = round(3.5)
print(e,f,g,h,i,j)
>>>
3.0 3 3 4 3 4
```

Em e o valor 3.9 foi convertido para 3 e depois para float 3.0. Em f o 3.9 float é convertido para o int 3. Em h o valor 3.9 é arredondado para o inteiro mais próximo que é 4. Em i o valor 3.49 foi arredondado o int mais próximo 3. Em j o valor 3.5 é arredondado para o 4, e não para o 3. O comando print() imprime os valores das variáveis.

BOX Conceito: Uma **função** é código computacional que executa uma funcionalidade. Abaixo temos uma definição matemática de função y=f(x). O x pertence ao dominio A e o y a imagem B. Na matemática, nas funções, imagens, domínios e variáveis são tipicamente nomeadas por uma letras, ex. A, B, x, y, f, g. Na programação preferimos nomes mais significativos. Tipo Inteiros (int) e reais (float), round, trunc, etc.



2.4 Expressões aritméticas

Como em outras linguagens temos vários tipos de operadores aritméticos, (+ - * /). Na divisão temos três operadores: / divisão float; // parte inteira da divisão; e % o resto da divisão inteira.

```
>>> 1/3
0.3333333333333333
>>> 5//2
2
>>> 5%2
1
```

Aritméticos	Parte e resto da divisão
+ - * / **	// (parte inteira)

	% (resto da divisão)

Operadores aritméticos

Além dos 4 operadores aritméticos básicos + - * /, temos a combinação deles com o operador de atribuição. Por exemplo, a+=1 é o mesmo que a=a+1 e a*=2 é o mesmo que a=a*2. O operador ** representa a potênciação: 2**4 = 2*2*2*2 = 16.

2.4.1 Praticando o código

Seguem alguns exemplos de expressões para serem praticadas no shell, como comandos individualizados.

```
>>> 2**3
8
>>> 10 / 3
3.3333333333333335
>>>10//3
3

>>> 10 % 3
1
>>> 8 / 3
2.6666666666666665
>>> 8 % 3
2
```

2.4.2 Prioridade dos operadores aritméticos

Na expressão 10*2**3 mesmo não colocando parênteses a parentização implícita fica 10*(2**3).

```
>>> 10*2**3
80
>>> (10*2)**3
8000
>>> 10*(2**3)
80
```

2.5 Lista de exercícios para o shell (3)

Execute estes comandos no python shell do IDLE. No Shell não é necessário salvar os comandos como arquivos de programas. Usa-se o python como uma calculadora. No shell para saber o valor de uma variável escreve-se: >>>x=4*5;x ou >>>x=4*5; >>>x.

E2.1	Qual é o valor de x nas expressões:
	
	`>>> x= 2**3`
	`( ) 0  ( )6  ( )8  ( )9  ( )27`
	
	`>>> x= 2 % 3`
	`( ) 0  ( )1  ( )2  ( )3  ( )1.5`

	>>> x= float(20 % 3)
	( ) 0 ( )1.0 ( )2 ( )2.0 ( )1.5
	
	>>> x= int(20 / 3)
	( )6 ( )6.0 ( )6.666 ( )3.333 ( )1.5
	
	>>> x= 3 - 2 * 4
	( ) 0 ( )4 ( )-5 ( )4.0 ( )-2
	
	>>> x= 3 * 2 ** 2
	( )0 ( )12 ( )36 ( )7 ( )100

E2.2	Converta para Python as expressões aritméticas: 10+2x30, 4^2+30, (9^4+2)x6-1
	
	>>> 1+2*3
	
	>>> 2**2+3*4
	
	>>> (1**2+2**2)*3-2

E2.3	Converta para Python as expressões aritméticas: a^2+5b+c, (a^3+2b)+c, onde a=7, b=100, c=1.

2.5.1 Lista de exercícios para o IDLE (11)

Estes códigos devem ser salvos como arquivos de programas individuais na IDLE.

E2.4	Seja um salário (sal) de 1000. Como dar um aumento de 15%. Qual o valor do aumento e do novo salário. sal=1000 percent=15 aumento=sal*percent/100 novosal=sal+aumento print(aumento, novosal) >>> 150.0 1150.0
E2.5	Seja um salário (sal) de 1500. Como dar um aumento de 20%. Qual o valor do aumento e do novo salário. sal=1500 percent=20 aumento=sal*percent/100 novosal=sal+aumento print(aumento, novosal) >>> 300 1800
E2.6	Seja um salário (sal) de 1500. Como dar um aumento de 10%. Qual o valor do aumento e do novo salário.

E2.7	Seja um salário (sal) de 1800. Como dar um aumento de 1%. Qual o valor do aumento e do novo salário.
E2.8	Escreva o resultado de 2a + 5b. Onde a=3 e b=5.
E2.9	Escreva o resultado de 2a x 5b. Onde a=7 e b=11.
E2.10	Escreva o resultado de 4a x -5b. Onde a=3 e b=5.

E2.11	Calcule a soma de três variáveis, a=3;b=5;c=11.
	
	`a=3`
	`b=5`
	`c=11`
	`abc=a+b+c`
	`print(abc)`
	`>>>`
	`19`

E2.12	Seja uma temperatura de 30 graus Celsius; converter para Fahrenheit: F=(9*C)/5+32.
E2.13	Sejam 28 dias. Quantas horas e quantos minutos tem em 28 dias? Por exemplo, 2 dias, são 48horas e 48*60=2880 minutos.
E2.14	Um senhor deixou um testamento para 3 filhas (A,B,C) e um filho (D). No testamento diz que as filhas A,B,C receberão respectivamente, 1/4, 5/16 e 3/8 partes e o filho homem receberá o restante. O valor que ele deixou é 999000. Imprima quanto cada um recebe.

2.5.2 Perguntas conceituais (7)

	Responda com no mínimo 10 palavras e no máximo 20 palavras:
C2.1	Quais os tipos de operadores associados a divisão? Qual a diferença entre eles?
C2.2	O que é uma variável? Variáveis armazenam objetos computacionais: inteiros, strings, etc. São usadas dentro de expressões e/ou em comandos.
C2.3	O que é um comando?
C2.4	O que é um algoritmo?
C2.5	O que é uma função?
C2.6	O que é um programa?
C2.7	Qual a diferença entre bit e byte? Um bit é a unidade mínima de informação, pode assumir os valores 0 ou 1; sequencias de bits formam números; um byte são oito bits.

2.6 Nerd - tópico opcional: INT, Bit e Byte

Este tópico é opcional, podendo ser pulado numa primeira leitura. Mas em computação falamos de bits e bytes então vamos examinar o que são bits e bytes. Valores int podem ser representados em diferentes bases. Pessoas perguntam qual é o tamanho de um inteiro em bytes. Podemos utilizar sys.getsizeof(73) para checar o tamanho em bytes ocupado na memória pelo número inteiro 73, que são 14 bytes.

```
>>> import sys
>>> sys.getsizeof(73)
14
```

BOX: Um byte são oito bits. O valor binário do inteiro em hexa 0xFF é um byte '0b11111111', que é o valor positivo 255. Num byte podemos representar a faixa de 0 até 255, que são 2**8=256 valores.

```
>>>bin(0xF)
'0b1111'
>>>bin(0xFF)
'0b11111111'
>>>bin(0xF0)
'0b11110000'
```

Em 32 bits podemos representar inteiros grandes. Por exemplo, int(0xF0080419)= 4.027.057.177, 4 bilhões, 27 milhoes, etc. O valor máximo é 2**32-1, que é 4.294.967.295.

```
>>>int(0xF0080419)
4027057177
>>>bin(0xF0080419)
'0b11110000000010000000010000011001'
```

Normalmente usa-se valores decimais (de base 10), mas pode-se definir valores octais (de base 8) e hexadecimais (base 16) ou binários (base 2). Para declarar um inteiro octal, hexadecimal ou binário usa-se os prefixos, 0o, 0x, 0b. Acima temos um hexa de oito dígitos 0xF0080419 que corresponde a 32 bits.

```
a = 42   #decimal
b = 0o10 #octal
c = 0xA1 #hexadecimal
d = 0b1010101
print(a,b,c,d)
>>>>
42 8 161 85
```

Podemos também converter um valor decimal para as diferentes bases usando-se as funções bin(), oct() e hex(). Os dígitos binários são utilizados na representação internas dos computadores, nas arquiteturas dos computadores. As operações internas na arquitetura dependem de portas lógicas que operam sobre bits e bytes. A unidade mínima de informação é o bit. Oito bits formam o byte. Alguns computadores possuem a memória formada por palavras de 8 bytes, ou 64 bits. Na escrita destes bytes é mais fácil utilizar a notação hexadecimal, por exemplo, dois dígitos hexa correspondem a um byte ou 8 bits.

```
dec = 344
print("O decimal",dec,"é:")
print(bin(dec),"em binario.")
print(oct(dec),"em octal.")
print(hex(dec),"em hexadecimal.")
>>>
O decimal 344 é:
0b101011000 em binario.
0o530 em octal.
0x158 em hexadecimal.
```

2.6.1 Tipo (type) da variável

A função type() permite testar o tipo de dados de um valor ou de uma variável. Por exemplo,

```
>>> type(10)
<class 'int'>
>>> type(12.1)
<class 'float'>
```

Neste caso cada tipo é visto como uma classe de objetos, deste ponto de vista vem a palavra class. Assim os inteiros, os float e os boolean, são também objetos.

Inteiros são simples e óbvios, mas Float representa números reais com ou sem expoente (e ou E). Esses números são comumente chamados de floating-point ou números de ponto flutuante. Por exemplo: 0.0042, .005, 1.14159265 e 6.02e-23 (o mesmo que 6.02×10^{-23}).

Executando os prints abaixo, temos o tipo de dados dos valores numéricos testados, diferenciando int de float.

```
print(type(5))
print(type(5.0))
print(type(4.3))
print(type(-2))
print(type(100))
print(type(1.333))
>>>
<class 'int'>
<class 'float'>
<class 'float'>
<class 'int'>
<class 'int'>
<class 'float'>
```

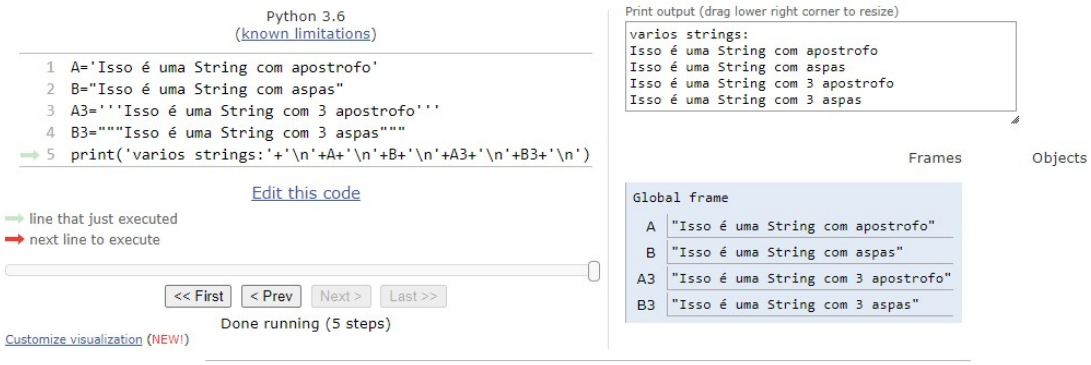
Capítulo
3 TIPO DE DADOS STRING

*Este capítulo apresenta o tipo de dados string usado para representar nomes e textos. Eles são muitos usados nos comandos de entrada e saída. Aqui tratamos de vários operadores e métodos de manipulação de strings: operadores + % *; métodos lower(), upper(), split(), strip(). Fatiamento de strings. Composição de strings. Caracteres ascii.*

3.1 Tipo STRING

Para atribuirmos para uma variável um string, basta que coloquemos entre aspas simples, ou triplas, como mostra o exemplo abaixo; pode ser também apóstrofo. Neste print concatenamos as 3 strings usando o operador +. Entre eles colocamos o string fim de linha "\n".

```
A='Isso é uma String com apostrofo'
B="Isso é uma String com aspas"
A3='''Isso é uma String com 3 apostrofo'''
B3="""Isso é uma String com 3 aspas"""
print('varios strings:'+'\n'+A+'\n'+B+'\n'+A3+'\n'+B3+'\n')
```



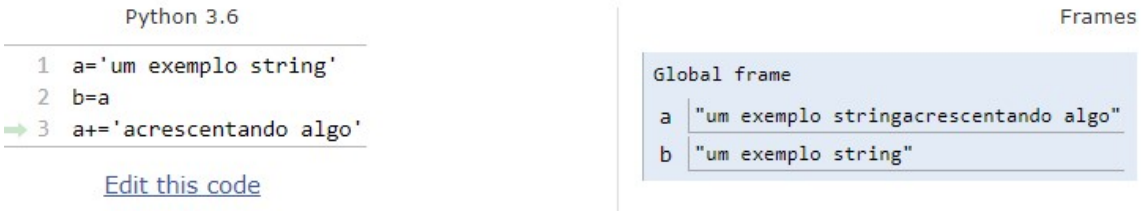
Strings no início de uma linha, sem o comando de atribuição são considerados comentários. O string com três aspas pode ocupar várias linhas. É útil em programas grandes para colocar uma parte do código programa como comentário, ou para representar um texto grande dentro do código do programa.

```
""" aqui é comentário
Exemplo com string
grande"""
```

```
b="""Este texto tem
algumas linhas
esta é a terceira
esta é a quarta"""
print(b.split())
>>>
['Este', 'texto', 'tem', 'algumas', 'linhas', 'esta', 'é', 'a',
'terceira', 'esta', 'é', 'a', 'quarta']
```

O método Split quebra o string e devolve uma lista, acima temos uma lista de palavras. Litas só veremos mais adiante, mas aqui vamos nos familiarizando com a notação.

Os strings em python são não mutáveis. Portanto, se temos uma string a fazemos b=a; serão duas cópias diferentes; se modificamos o a não modifica o b. Para isso acontecer as variáveis string são referencias (apontam para) os conteúdos. Ver abaixo.



3.2 Composição de string

O tipo string permite muitas operações, entre elas, atribuição =, concatenação +, multiplicação *, composição %. Nas linhas 1 e 2, abaixo, são criadas 2 variáveis as quais atribuímos uma string. Na linha 3 usa-se a composição de strings %, um recurso que preenche **os marcadores escritos com %s**, do tipo string, a partir de uma tupla de valores (valores entre parênteses).

```
nome1 = "Linguagem Python"
nome2 = 'Algoritmos'
print ("Texto de %s para programação na %s" % (nome2, nome1))
>>>
Texto de Algoritmos para programação na Linguagem Python


>>> '-'*10
'----------'
```

Para a **composição de strings** temos inúmeros marcadores, para os diferentes tipos de dados: caracteres, string, decimais com sinal e sem sinal, decimais em outras bases (ver final do capítulo 2) e ponto flutuante.

Marcador do tipo de dados	Conversão
%s	String
%i, %d	Decimal inteiro
%x, %X	Hexadecimal Inteiro
%e, %E	Notação exponencial
%f	Ponto flutuante (Números reais)

Alguns formatos para a composição de strings

```
>>> 'tempo: %5.2f horas' % (42.232232)
'tempo: 42.23 horas'
```

No exemplo acima o tempo é mostrado apenas com duas casas decimais. 5 casas no total, duas depois do ponto. Aos poucos vamos introduzindo o uso destes formatos para a composição de strings.
Além dos operadores (+ % *) para strings temos vários outros métodos tais como lower(), upper(), split(), strip(), etc. O método startswith(prefixo) retorna True se uma string começa com o prefixo passado por argumento, e False caso contrário.
O termos lower(), upper(), split(), strip() significam:

- lower(), caixa baixa
- upper(), caixa alta
- split(), quebra
- strip(), limpa brancos
- startswith(), começa com
- len(), (length), comprimento
- capitalize(), primeira letra em maiúscula

Seguem alguns exemplo. Como trabalhar com letras maiúsculas e ou minúsculas: O capitalize(), exibe em caixa alta apenas o primeiro caractere de cada palavra; o upper() retorna em caixa alta e lower() retorna em caixa baixa.

```
>>> nome1 = "Linguagem Python"
>>> nome1.upper()
'LINGUAGEM PYTHON'
>>> nome1.lower()
'linguagem python'
```

Podemos testar como começa o início de um string, no caso o nome1 definido acima. Depois o método strip() retorna uma cópia da string com os espaços em branco removidos (antes e depois).

```
>>> nome1.startswith('lin')
False
>>> nome1.startswith('Lin')
True

>>> ' aaa '.strip()
'aaa'
```

Duas funções muito úteis são split() e len(). A Split() quebra e retorna uma lista com os pedaços. A função len() determina o comprimento da string (quantas letras possui).

```
>>> x='2,5,7, 12, 15, 7'
>>> x.split(',')
['2', '5', '7', ' 12', ' 15', ' 7']
>>>
>>> len('abcd')
4
```

3.3 Indexação e fatiamento

Abaixo temos um exemplo de **fatiamento** de string [:], [início:fim]. nome1[10:] pega a partir da posição 10 até o fim; nome[10:13] pega a partir da posição 10 até a 12; nome[:5] pega do início até o 4. Note que o início conta mas o fim não conta, é exclusive fim: fim=13 na prática é até 12; fim=5 na prática é até 4. Isto é porque **a indexação (numeração) começa no zero**. Na prática nome1[:5] pega os primeiros 5 caracteres.

```
>>> nome1 = "Linguagem Python"
#            0123456789012345
>>> nome1[10:]
'Python'
>>> nome1[10:13]
'Pyt'
>>> nome1[:5]
'Lingu'
```

3.4 Chr() & Ord() - Carateres ASCII

A tabela Ascii define o mapeamento do conjunto de caracteres para o valor inteiro representável num byte. As funções chr() e ord() codificam o mapeamento. A função chr() converte o valor inteiro num caractere e a função ord() retorna o valor inteiro do caractere.

```
>>> chr(48)
'0'
>>> chr(49)
'1'
>>> chr(65)
'A'
>>> chr(90)
'Z'
>>> chr(97)
'a'
>>> ord('3')
51
>>> ord('b')
98
```

Abaixo temos um resumo da **tabela Ascii**, para os principais grupos de letras. As letras de controle são utilizadas internamente pelos sistemas computacionais principalmente para comunicação de dados. O ord(' ')=32; ord('\t')=9; ord('\n')=10; respectivamente ord() do espaço, tabulação e nova linha.

Tipo	Valores	Representação
controle	0..8, 14..31, 127	
alfabéticos	65..90, 97..122	ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
dígitos	48..57	0123456789
espaços	9..13, 32	
pontuação	33..47, 58..64, 91..96, 123..126	!"#$%&'()*+,-./ :;<=>?@ [\]^_` {|}~

3.5 Lista de exercícios (8)

Programe os exercícios utilizando fatiamento mais as funções e operadores sobre string apresentados acima.

E3.1	Seja, a='jhon lenon' . Imprima o conteúdo da variável com ambos os nomes capitalizados. Use capitalize().
E3.2	Seja, a='jhon lenon' . Imprima o conteúdo da variável todo em caixa alta. Use upper().

E3.3	Seja, a='jhon lenon' . A partir de a, imprima "Lenon, Jhon". Use fatiamento e capitalize().
	`a='jhon lenon'`
	`#  0123456789`
	`print(a[5:].capitalize()+ ', ' + a[:4].capitalize())`
	`>>>`
	`Lenon, Jhon`

E3.4	Seja, a='jhon lenon' . A partir de a, imprima "Jhon, Jhon, Jhon, Lenon. Use fatiamento capilatize() e também o operador * para string.
E3.5	Seja, a='jhon lenon' . A partir de a, imprima "----------Jhon---------". Use fatiamento e também o operador * para string.

E3.6	Seja, a='jhon lenon' . A partir de a, imprima "J. Lenon".
	
	`a='jhon lenon'`
	`#  0123456789`
	`print(a[0].upper()+ '. ' + a[5:].capitalize())`
	`>>>`
	`J. Lenon`

E3.7	Seja b="tinoco". Escreva `"TINoco"`, pegando fatias da string b e upper()
E3.8	Seja a='jhon \nlenon \n'. Escreva `"LENON, JHON"`, pegando fatias da string a e upper(). Cuidado, na contagem dos chars em 'jhon \nlenon \n' temos apenas 13 chars e não 15, pois cada '\n' conta 1.

3.5.1 Perguntas conceituais (4)

Responda com no mínimo 10 palavras e no máximo 20 palavras:

C3.1	Qual a diferença entre uma string declarada com aspas (apóstrofos) ou 3 aspas?
C3.2	Quais os diferentes usos do operador %, para inteiros e para strings ?
C3.3	Qual o uso do “\n”?

RESUMO:
string com um apóstrofo ou aspas
string com três apóstrofos ou aspas (multi linhas)
string no início de linha como comentário
operadores sobre expressões de string: concatenação +, multiplicação *, composição %
lower(), caixa baixa
upper(), caixa alta
split(), quebra
strip(), limpa brancos
startswith(), começa com
len(), (length), comprimento
capitalize(), primeira letra em maiúscula
fatiamento [início:fim]
tabela ascii
ord() valor ascii do caractere
chr() char do valor ascii

3.6 Nerd: Int para string & string para Int (OPCIONAL NUM COMEÇO)

Por exemplo, '1312'= (((((1*10)+3)*10)+1)*10)+2. Então para pegar o valor 1 a partir do '1' devemos fazer ord('1')-ord('0')=49-48=1, pois, em Ascii os números são sequênciais; o valor 7 a partir de '7' devemos fazer ord('7')-ord('0')=55-48=7.

```
def str2int(S):
    val=0; LEN=len(S)
    for i in range(LEN):
        dig=ord(S[i])-ord('0')
        val = val*10+dig
    return val
print(str2int('1312'), str2int('0'), str2int('100'))
print(int('1312'), int('0'), int('100'))

>>>
1312 0 100
```

Este algoritmo srt2int() simula a função int() do Python, mas nossa versão ainda é limitada, pois o int() aceita [+-] o sinal na frete do número e aceita colocar caracteres de espaço antes e/ou depois. Por fim, também se pode trabalhar com bases, 2, 8, 16, e.g., na base 2, int('10101',2) = int('0b10101',2)=21.

```
>>> int(' +32  ')
32
>>> int(' -32  ')
-32
>>> int('10101',2)
21
```

A solução abaixo codifica o inverso, int para str, int2str(). Mas vamos fazer sem usar str(). Para um só dígito é simples: str(D) é equivalente a chr(D+ord('0')), por exemplo, chr(7+ord('0'))='7' e str(7)='7'. Para converter um inteiro, >9, numa sequência de dígitos fazemos: enquando for maior ou igual a 10, basta pegar sempre o resto da divisão inteira por 10. Por exemplo, a partir de 312, 312//10 da 31 e 312%10 dá **2**; depois a partir de 31, 31 // 10 dá 3 e 31%10 dá **1**; depois 3<10, pega-se o **3**. Juntando tudo da [2,1,3]. Revertendo da [3,1,2].

```
def dig(x): return chr(x+ord('0'))
def int2str(I):
    STR=[]
    while I>=10:
        STR.append( dig(I%10) )
        I = I//10
    else: STR.append( dig(I) )
    return ''.join(reversed(STR))
print(int2str(312), int2str (0), int2str(10000))
>>>
312 0 10000
```

Este algoritmo facilmente pode manipular inteiros negativos, basta guardar o sinal do número, neg=x<0;x=abs(x), e trabalhar com o valor abs(). No final, se for negativo acrescentar o sinal no início da string.
Está solução também pode ser generalizada para outras bases, substituído-se o 10 pela base, por exemplo, 2, 8. Para a base 16 a função dig() deve ser ajustada para trabalhar com os dígitos “0123456789abcdef”.

3.7 Exercícos sobre chr e ord *COMPLEXO

E3.1 Faça a função srt2int(), string para inteiro. '1312'= (((((1*10)+3)*10)+1)*10)+2. Para criar o valor 7 a partir de '7' devemos fazer ord('7')-ord('0')=55-48=7.

```
def str2int(S):
    val=0; LEN=len(S)
    for i in range(LEN):
        dig=ord(S[i])-ord('0')
        val = val+dig
        if i<LEN-1: val*=10
    return val
print(str2int('1312'), str2int('0'), str2int('100'))
>>>
1312 0 100
```

E3.2 Faça a função dig() que retorna o valor inteiro de um digito char

```
def dig(x): return chr(x+ord('0'))
```

E3.3 Faça a função int2str que transforma um inteiro para string usando dig()

```
def int2str(I):
    STR=[]
    while I>=10:
        STR.append( dig(I%10) )
        I = I//10
    else: STR.append( dig(I) )
    return ''.join(reversed(STR))
print(int2str(312), int2str (0), int2str(10000))
>>>
312 0 10000
```

E3.4 Faça a função int2str generalizada para trabalhar com hexadecimais. Está solução também pode ser generalizada para outras bases, substituído-se o 10 pela base, por exemplo, para 16 a função dig() deve ser ajustada para trabalhar com os dígitos “0123456789abcdef”.

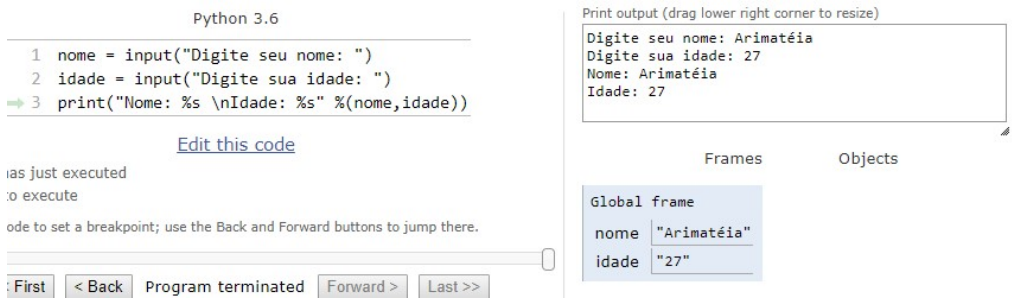
Capítulo

4 COMANDOS DE ENTRADA E SAÍDA

Este capítulo apresenta os comandos básicos de entrada e saída do python: input e print.

4.1 Entrada de Dados

Em Python, a leitura de dados do teclado é feita através da função input(), onde se pode colocar um texto. O print() escreve na saída.



Ler duas variáveis a e b do tipo int e imprimir a sua soma. Uma leitura em Python com o comando input() retorna uma string, por exemplo `'12'` que pode ser convertido em 12 pela função int(`'12'`). Assim o programa fica como:

```
a=int(input('Digite a:'))
b=int(input('Digite b:'))
ab=a+b
print('a+b=',ab)
```

Vamos agora ler uma velocidade e uma distância e imprimir o tempo necessário para percorrer a distância.

```
velo=int(input('Velocidade km/hora:'))
dist=int(input('Distância km      :'))
tempo = dist/velo
print('para percorrer %d km numa vel %d km/h precisa %5.2f horas'
      % (dist, velo, tempo))
>>>
Distância km      :500
Velocidade km/hora:70
```

```
para percorrer 500 km numa vel 70 km/h precisa  7.14 horas
```

4.2 Leitura de Float

Toda leitura lê um string, para converter o string para float usa-se a função float(). Como segue. De forma similar foi exemplificado acima para int.

```
velo=float(input('Velocidade km/hora:'))
dist=float(input('Distância km      :'))
tempo = dist/velo
print('para percorrer %d km numa vel %d km/h precisa %5.2f horas'
      % (dist, velo, tempo))
```

4.3 Leitura de strings

Toda leitura lê um string, portanto não precisa conversão.

```
nome=input('digite o Nome:')
print('O nome lido foi: ', nome)
```

4.4 Lista de exercícios (11)

E4.1	Ler um nome e escreve-lo capitalizado. Aqui, na solução usamos a variável nome diferente de Nome.
	
	`nome=input('digite o nome:')`
	`Nome=nome.capitalize()`
	`print(nome, Nome)`
	`>>>`
	`digite o nome:claudio`
	`claudio Claudio`

E4.2	Ler um nome e escreve-lo capitalizado. Aqui, na solução usamos as variáveis a e b.
	
	`a=input('digite o nome:')`
	`b=a.capitalize()`
	`print(a, b)`
	`>>>`
	`digite o nome:claudio`
	`claudio Claudio`

E4.3	Ler um nome e escreve-lo todo em caixa alta.
E4.4	Ler nome, identidade, cpf. Imprimir os dados lidos, um em cada linha.

E4.5	Ler um nome tipo “Paulo dos santos dos deuses”. Usando Split quebre no ‘dos’ em três partes.
	
	`b= input('digite o nome:')`
	
	`print(b.split('dos'))`

	>>> digite o nome: Paulo dos santos dos deuses ['Paulo ', ' santos ', ' deuses']
E4.6	Ler um nome e escrever ele e seu comprimento, use len().
E4.7	Sejam as cotações a) Dólar Turismo R$ 5,6000; b) Euro R$ 6,20; c) Libra R$ 6,9769 Peso Argentino d) R$ 0,0710; e) Bitcoin 10,4200. Isto é, para comprar 1 Dolar preciso pagar 5,6 reais e assim por diante. Leia um valor em real e converta o mesmo para as 5 moedas, imprimindo o valor em real e o correspondente em cada uma das moedas, caso fosse cambiar o real por uma das moedas.
E4.8	Ler um salário (sal). Dar um aumento de 20%. Qual o valor do aumento e do novo salário. sal=int(input(' digite o sal:')) percent=20 aumento=sal*percent/100 novosal=sal+aumento print('sal:', sal, 'aumento:',aumento, 'novosal:', novosal) >>> digite o sal:2000 sal: 2000 aumento: 400.0 novosal: 2400.0
E4.9	Ler um salário (sal). Dar um aumento de 2%. Qual o valor do sal, do aumento e do novo salário.
E4.10	Considere a fórmula: C/5 = (F-32)/9 para conversão de temperatura. Programe a conversão de Celsius para Fahrenheit. Leia um valor em Celsius e escreva o correspondente em Fahrenheit.
E4.11	Considere que uma polegada é 2,54cm. Leia um valor em cm e escreva o correspondente em polegadas.

Capítulo
5 EXPRESSÕES LÓGICAS

Este capítulo apresenta as expressões lógicas como uma preparação para vermos os comandos condicionais e de repetição. São apresentadas as tabelas verdade para os operadores lógicos: and, or, not. São apresentadas as expressões relacionais de igualdade, desigualdade e pertinência.

5.1 Tipo de dados BOOL

O nome Bool vem de George Boole (1815-1864), matemático inglês, criador da "Álgebra Booleana" de onde vem a lógica proposicional ou boleana, a lógica das tabelas verdade. É a lógica do pensamento humano. Sempre utilizamos frases em linguagem natural com (e), (ou) e (não).
Por exemplo, se dizemos “compre alho e cebola”. Neste caso queremos dizer “compre alho e também cebola”.
Se dizemos “tempere com sal ou com açúcar”. Queremos dizer “tempere com um dos dois: ou sal, ou açúcar”.
Podemos fazer frases em linguagem natural com vários conectivos lógicos: “vamos pescar se faz sol e não venta ou se chove e está quente”. Aqui temos quatro conectivos lógicos: e, não, ou, e.
O **tipo lógico bool** é uma especialização do tipo int. Os valores do tipo bool são: True (int diferente de 0) e False (int igual a 0). True é verdadeiro e False é falso. **CUIDADO** True é com T maiúsculo e False é com F maiúsculo, senão dá erro. bool() é a função que converte inteiro para Bool.

```
>>> bool(1)
True
>>> bool(-10)
True
>>> bool(10)
True
>>> bool(0)
False
>>> a=True
>>> a and bool(2)
True
>>> True + 1
2
>>> False + 1
1
>>> True + True + False
2
>>> true+1
```

```
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module> true+1
NameError: name 'true' is not defined
>>>
>>> Bool(0)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module> Bool(0)
NameError: name 'Bool' is not defined
```

5.2 Operadores lógicos

Os três operadores lógicos da linguagem natural (não, e, ou) são transportados para o pensamento computacional com os operadores lógicos: not, and, or.

Operadores Lógicos
not and or

Estes três operadores podem ser combinados para fazer expressões lógica complexas, seguem alguns exemplos.

```
>>> True and False
False
>>> True and True
True
>>> True and True and not True
False
>>> True or False and True
True
>>>
```

5.2.1 Tabelas Verdade

Com o auxilio das tabelas verdade, podemos avaliar as expressões lógicas, reduzindo uma expressão num valor final que será verdadeiro ou falso.

Tabelas Verdades:

not	and	or
not True = False **not False = True**	**True and True = True** **True and False = False** **False and True = False** **False and False = False**	**True or True = True** **True or False = True** **False or True = True** **False or False = False**

O not é um operador unário, ele nega o valor. O and é um operador binário, ocorre entre dois operandos. O and é sempre False exceto quando o valor dos dois operandos for True

então neste caso o valor dele também é True. O or é sempre True exceto quando o valor dos dois operandos for False então neste caso o valor dele também é False.

Com estas tabelas podemos resolver expressões do tipo "((not A) and B) or C". Se A= B=C=True. O resultado será:
((not True) and True) or True =
(False and True) or True =
False or True =
True

Agora se A=B=C=False, o resultado será
((not False) and False) or False=
(True and False) or False=
False or False=
False

Agora se A=C=False, B=True, o resultado será:
((not False) and True) or False=
(True and True) or False=
True or False=
True

5.2.2 Prioridade na parentização implícita

Se temos uma expressão "not A and B or C" sem parenteses, como ela será avaliada? Por exemplo, se A=False; B= True; C=True. A parentização correta é ((not A) and B) or C. Primeiro o not, depois o and e depois o or, nesta ordem.
Fazendo um paralelo com a aritmética, -A*B+C é parentizado de forma similar((-A)*B)+C. Se for B+C* -A será parentizada B+(C*(-A)). Assim, B or C and not A fica B or (C and (not A)).

```
>>> A=False; B= True; C=True
>>> not A and B or C
True
>>> not (A and B or C)
False
>>> (not A) and (B or C)
True
>>> ((not A) and B) or C
True
```

Quando temos múltiplas atribuições numa linha podemos também utilizar o comando de **atribuição múltipla**, veja o código abaixo:

```
>>> A,B,C=False,True,True # como no exemplo acima
>>> not A and B or C
True
```

5.2.3 Impressão das tabelas verdade

Para nos familiarizar com as tabelas, podemos imprimi-las. Vamos apresentar um pseudo código para imprimir a tabela verdade do not.

Em pseudo-código (código que não executa):

```
escreva('Tabela Verdade not:')

VF = [v-verdadeiro, f-falso]
para a em VF:
     escreva('not',  a, '=', not a)

>>>
Tabela Verdade not:
not True = False
not False = True
```

O pseudo código serve como uma notação informal para representar algoritmos, baseado numa linguagem próxima da linguagem natural. A atribuição VF = [v-verdadeiro, f-falso] cria ula lista com duas opções uma é o verdadeiro a outra é o falso. “para a em VF” é uma expressão de pertinência: “para a pertencendo a um dos valores de VF”.

Este pseudo código pode ser traduzido para uma linguagem de programação que para nós é Python.

Em Python:

```
print('Tabela Verdade not:')
TF = [True, False]
for a in TF:
     print( 'not',  a,'=', not a)
>>>

Tabela Verdade not:
not True = False
not False = True
```

De forma similar podemos imprimir a tabela verdade do and e do or. Vamos agora fazer um pseudo-código com duas variáveis a e b, que podem assumir os valores verdadeiro e falso. Este comando “para a em VF: para b em VF” permite a combinação duas a duas das variáveis a e b.

Em pseudo-código:

```
escreva('Tabela Verdade and:')

VF = [v-verdadeiro, f-falso]
para a em VF:
   para b em VF:
     escreva(a, 'and',  b, '=', a and b)

>>>
Tabela Verdade and:
True and True = True
```

```
True and False = False
False and True = False
False and False = False
```

Este pseudo código é diretamente traduzido para a linguagem python. De forma similar podemos imprimir a tabela do or.

```
print('Tabela Verdade and:')
TF = [True, False]
for a in TF:
    for b in TF:
        print(   a,'and', b, '=', a and b)
>>>
Tabela Verdade and:
True and True = True
True and False = False
False and True = False
False and False = False


print('Tabela Verdade or:')
TF = [True, False]
for a in TF:
    for b in TF:
        print(   a,'or', b, '=', a or b)
>>>
Tabela Verdade or:
True or True = True
True or False = True
False or True = True
False or False = False
```

5.3 Expressões Relacionais

As expressões relacionais também retornam o tipo de dados lógico, ou bool. Os operadores relacionais comparam valores numa relação de ordem, onde podemos testar se eles são iguais ou se um é maior ou menor que o outro. Além disso, em Python também temos uma comparção de pertinência, que permite testar se um valor pertence ou não a uma lista ou conjunto. Assim, temos 3 tipos de operadores relacionais: de igualdade, de desilgualdades e de pertinência.

Igualdade	Desigualdade	Pertinência
== !=	> < >= <=	in not in

5.3.1 Operadores relacionais: quando o igual não é igual

Os operadores lógicos normalmente são combinados em expressões com operadores relacionais, operadores que comparam valores, como o maior, menor e igual, etc. > < == != <= >=.

Muitas linguagens de programação usam o sinal de igual = para atribuição e == para teste de igualde. Porque a atribuição é utilizada com muita mais frequência que o teste de igualdade.

Se fizermos a=1;b=2;a=b. O comando a=b não é teste de igualdade: o valor de a é substituído pelo valor de b. Aqui o (;) é um separador de comandos, quando mais de um comando são escritos na mesma linha.

O diferente é o não igual, representado pelo ponto de exclamação na frente do igual. Assim, == e != ambos tem dois caracteres, assim ficam símbolos mais similares, no tamanho.

```
>>> a=1;b=2
>>> a==b
False
>>> a==1
True
>>> a==2
False
>>> a!=2
True
```

5.3.2 Prioridade dos tipos de operadores

Na expressão prova1>7 and prova2>7 and prova3>7 temos operadores relacionais e lógicos. Neste caso os operadores relacionais tem prioridade, são executas antes dos lógicos. Então esta expressão com os parênteses fica: (prova1>7) and (prova2>7) and (prova3>7). Se temos operadores aritméticos e relacionais, por exemplo, em x+7>4 os operadores aritméticos tem prioridade então com parênteses fica: (x+7)>4.

```
>>> prova1=8;prova2=9;prova3=9.5
>>> prova1>7 and prova2>7 and prova3>7
True
>>> (prova1>7) and (prova2>7) and (prova3>7)
True
>>> x=3
>>> x+7>4
True
>>> (x+7)>4
True
```

Os operadores de comparação ou relacionais são parentes dos aritméticos. Os aritméticos sempre retornam um valor numérico e os relacionais sempre retornam um valor Lógico. Seguem alguns exemplos, de expressões agrupadas num código de programa.

```
a=4;b=2;c=10;d=5;f=4
print('a==b',a==b)
print('a!=b',a!=b)
print('a<b' ,a<b)
print('b>=a', b>=a)
print('b<=a', b<=a)
print('a==f',a==f)
print('c+1<d',c+1<d)
print('c<d+6',c<d+6)
print('c>=f+6',c>=f+6)
print('f>=c',f>=c)
```

```
print('f<=f<=f<=f<=f',f<=f<=f<=f<=f)
print('f<f+1<f+2',f<f+1<f+2)
>>>
```

5.3.3 Operadores relacionais encadeados

Verificar mais de uma condição é muito comum em linguagens de programação. Se queremos verificar a condição a <b <c o usual é fazer a<b e b<c. Em Python, há uma maneira melhor de escrever isso usando operadores relacionais encadeados da seguinte forma a<b<c.
De acordo com a **associatividade e precedência** em Python todas as operações de comparação em Python têm a mesma prioridade. Assim uma expressões como a<b<c têm a interpretação convencional do uso em matemática. A associatividade é da esquerda para a direita.

Uma comparação pode usar vários operadores encadeados, por exemplo, x<=y<=z<=w é equivalente a x<=y and y<=z and z<=w. Como a equivalência é a uma lista de operações relacionais conectadas por and, quando a primeira condição for falsa as demais condições não são testadas (shortcut evaluation).

```
>>> x = 3
>>> 1 < x < 9
True
>>> 10 < 3*x < 18
False
>>> x < 10 < x*10 < 50 > 12*x
True
>>> 9 > x <= 3
True
>>> 5 == x < 4
False
```

5.4 Lista de exercícios (8)

E5.1	Seja uma variável sal=900. Crie uma variável lógica que é verdadeira se sal>1000. Imprima o valor da variável.
	
	`sal=900`
	`salMaiorMil=sal>1000`
	`print('maior que mil: ', salMaiorMil)`
	`>>>`
	`maior que mil: False`

E5.2	Seja uma variável sal=2000. Crie uma variável lógica que é verdadeira se sal>1000. Imprima o valor da variável.

E5.3	Crie uma variável aprovado que é verdadeira se todas as três provas (prova1, prova2, prova3) são maiores que 7. Inicialize as variáveis com valor maior que sete.
	
	`prova1=8; prova2=9; prova3=9.5`
	`aprovado= prova1>7 and prova2>7 and prova3>7`
	`print(aprovado)`

	>>> True
E5.4	Crie uma variável aprovado que é verdadeira se todas as três provas (prova1, prova2, prova3) são maiores que 7. Inicialize uma das provas com valor 6 e as outras com valor > 7. Escreva o valor da variável aprovado.
E5.5	Seja a atribuição múltipla: A,B,C,D=1,2,True,False. Mostre o valor da expressão 'A>B and C or D' A,B,C,D=1,2,True,False print('A>B and C or D =', A>B and C or D) >>> A>B and C or D = False
E5.6	Seja a atribuição múltipla: A,B,C,D=10,3,False,False. Mostre o valor da expressão 'A>B and C or D'
E5.7	Seja a atribuição múltipla: A,B,C,D=5,1,True,True. Mostre o valor da expressão 'A>B and C or D' A,B,C,D=5,1,True,True print('A>B and C or D =', A>B and C or D) >>> A>B and C or D = True
E5.8	Seja a atribuição múltipla: A,B,C,D=5,1,False,False. Mostre o valor da expressão 'A>B and C or D'

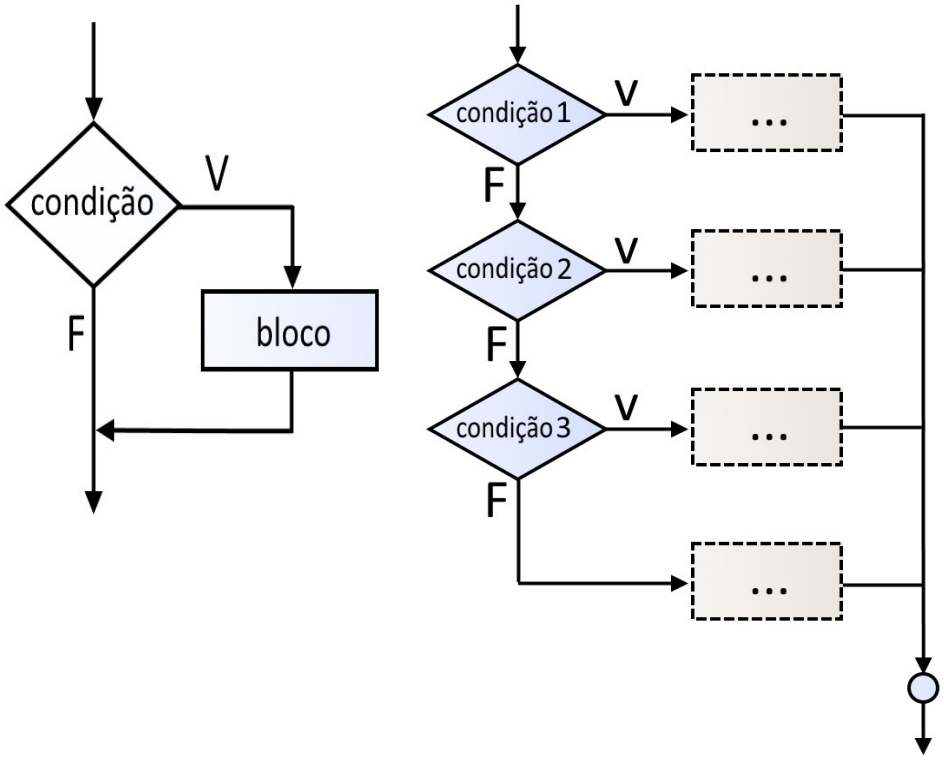
5.4.1 Perguntas conceituais (8)

	Responda com no mínimo 10 palavras e no máximo 20 palavras:
C23.1	Quais são os três principais operadores lógicos?
C23.2	O que retorna uma expressão lógica?
C23.3	Qual a diferença = e ==?
C23.4	Qual a diferença de um operador de igualdade e de desigualdade?
C23.5	Quais são os operadores de igualdade?
C23.6	Quais são os operadores de desigualdade?
C23.7	Quais são os operadores de pertinência?
C23.8	O que é um operador relacional?

Capítulo

6 COMANDOS DE DECISÃO (IF)

Neste capítulo apresentamos os comandos de decisão que permitem fazer escolhas condicionais sobre a execução de blocos de comandos. São variações sobre o comando if/elif/else. Fluxogramas dos comandos condicionais. Pseudo código de comandos condicionais.



Diagramas: a) um comando if simples *b) vários comandos ifs em cascata*

A figura acima ilustra dois **fluxogramas** do comando if. Um fluxograma representa um fluxo de execução da sequência de passos de um algoritmo. Um comando if simples, executa

condicionamente o bloco, isto é, se a condição for verdadeira então o bloco é executado, caso contrário ele não é executado. Quando temos ifs em cascata, todos os blocos com a condição verdadeira são executados.

6.1 Comando IF

Em Python temos vários usos do comando if/elif/else. Vamos ler um valor e dizer se é negativo, positivo ou igual a zero. Vamos usar três vezes o if simples, que resulta num if em cascata. Note que, no código executável, após a leitura do input() o valor é convertido para inteiro int().

Pseudo código comando se então:

```
x=ler('Digite x:')
se  x>0 então: escreva(x, 'é positivo')
se x==0 então: escreva(x, 'é zero')
se  x<0 então: escreva(x, 'é negativo')
```

```
x=int(input('Digite x:'))
if  x>0: print(x, 'é positivo')
if x==0: print(x, 'é zero')
if  x<0: print(x, 'é negativo')
>>>
Digite x:1
1 é positivo
```

Estes ifs estão em cascata, assim mais de um comando poderia ser executado. Todas as condições são testadas. Mas neste caso os três testes são mutualmente exclusivos, se um é verdade os outros dois não são. Então só um deles é executado. Este mesmo código pode ser escrito com if/elif, podem ser vários else/if=elif.

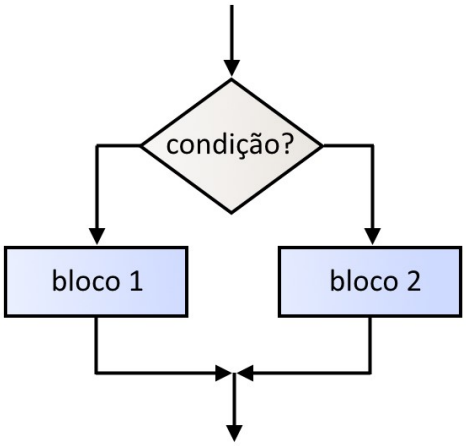
Pseudo código comando se então senão_se:

```
x=ler('Digite x:'))
se        x>0 então: escreva(x, 'é positivo')
senão_se x==0 então: escreva(x, 'é zero')
senão_se  x<0 então: escreva(x, 'é negativo')
```

```
x=int(input('Digite x:'))
if    x>0: print(x, 'é positivo')
elif x==0: print(x, 'é zero')
elif  x<0: print(x, 'é negativo')
```

A diferença desta versão para a versão em cascata é que aqui os testes são executados somente até a primeira condição ser verdadeira. Quando um if ou um elif for verdadeiro todos os que vem depois não são mais testados.

6.2 Comandos IF ELSE, aninhados



Fluxograma do comando if / else

O fluxograma do comando if/else executa uma ação quando a condição é verdadeia e outra ação quando a condição é falsa.

Pseudo código comando se então senão

```
x=ler('Digite x:'))
se x>=0
     então: escreva(x, 'é positivo')
     senão: escreva(x, 'é negativo)
```

```
x=int(input('Digite x:'))
if x>=0: print(x, 'é positivo')
else:    print(x, 'é negativo')
```

Novamente, abaixo, codificamos o mesmo algoritmo anterior, agora com dois if/else aninhados. Por aninhado entende-se um dentro do doutro. Note que não se pode escrever dois ifs na mesma linha. O segundo tem que ser indentado dentro do bloco do else. Um bloco é uma sequência de um ou mais comandos como uma unidade, dentro de uma indentação (vertical) ou dentro de uma linha (horizontal) sequênciados pelo ponto e virgula.

```
x=int(input('Digite x:'))
if x>0: print(x, 'é positivo')
else:
    if x==0: print(x, 'é zero')
    else :  print(x, 'é negativo')
```

Suponhamos agora uma conta de celular com diferentes taxas por minuto conforme o consumo total. Se o usuário aumenta o número de minutos consumidos o valor por minuto baixa, caindo numa faixa de valor por minuto mais baixa que vale para a conta toda.

```
min=int(input('digite o numero de minutos:'))
```

```
if   min>480: preco=0.18
elif min>240: preco=0.20
elif min>120: preco=0.25
else:         preco=0.30
print('minutos consumidos:',min)
print('preço por minutos :',preco)
print('valor total       :',preco*min)
>>>
digite o numero de minutos:250
minutos consumidos: 250
preço por minutos : 0.2
valor total       : 50.0
```

6.3 IFs mais complexos

Supondo que queremos formar uma conta de energia diferenciando os clientes em industriais, comerciais e residenciais, cada um com tarifas diferenciadas, inclusive variando por faixa de consumo.

Com	Acima de 5000kw preço 0.70 Abaixo/inclusive 5000kw preço 0.50
Ind	Acima de 10000kw preço 0.60 Abaixo/inclusive 10000kw preço 0.40
Res	Acima de 500kw preço 0.80 Abaixo/inclusive de 500kw acima de 200Kw preço 0.60 Abaixo/inclusive de 200kw preço 0.40

Interpretando esta tabela, vamos considerar que a tarifa baixa é para o valor Kw consumido dentro daquela faixa. Por exemplo, para um cliente residencial o consumo de 400Kw ele pagará 200 a 0.40 e 200 a 0.60, que resultará em 200,00.

```
kw=int(input('quantos Kwh:'))
tipo=  input(' tipo conta:')
if tipo not in ['r','i','c']:
    print('tipo de conta Invalido:',tipo);
    exit(1)
if tipo=='c':
   if kw > 5000: preço = (kw-5000)*0.70 + 5000*0.50
   else:    preço = kw*0.50
if tipo=='i':
   if kw > 10000: preço=(kw-10000)*0.60 + 10000 * 0.40
   else: preço=kw*0.40
if tipo=='r':
   if   kw > 5000: preço = (kw-500)*0.80 + 300*0.60 + 200*0.40
   elif kw >  200: preço =             (kw-200)*0.60+ 200*0.40
   else: preço = kw*0.40
print('pagar para %d valor = %5.2f' % (kw,preço))
>>>
quantos Kwh:400
 tipo conta:r
pagar para 400 valor = 200.00
```

Aqui temos mais algumas construções: o **exit(1), sair(1),** termina o programa. A expressão tipo **not in ['r','i','c']** checa se o tipo não é uma das três letras válidas; “not in” significa não pertence.

6.4 Exercícios (9)

Para cada exercício executar com pelo menos uma entrada de dados, para verificar se a resposta está correta.

E6.1	Ler A, B, C inteiros e imprimir o maior (pode ter mais de um maior).
	
	`a=int(input(' A:'));b=int(input(' B:'));c=int(input(' C:'))`
	`if a>=b and a>=c:`
	`  print('o maior é a:', a)`
	`if b>=a and b>=c:`
	`  print('o maior é b:', b)`
	`if c>=a and c>=b:`
	`  print('o maior é c:', c)`
	`>>>`
	` A:3`
	` B:7`
	` C:1`
	`o maior é b: 7`

E6.2	Ler A inteiro e dizer se A é par ou impar. Se o resto da divisão inteira de A por 2 for igual a zero então é o número par.

E6.3	Ler um valor de salário, sal, e se for maior que 1000 dar um aumento de 10% senão dar um aumento de 15%. Imprimir o valor do salário, do aumento e o novo salário.
	
	`sal=float(input('sal:'))`
	`if sal>1000:`
	`  percent=0.10`
	`else:`
	`  percent=0.15`
	`aumento=percent*sal`
	`novoSal=sal+aumento`
	`print('sal atual: R$%6.2f' % sal)`
	`print('  aumento: R$%6.2f' % aumento)`
	`print('novo sal : R$%6.2f' % novoSal)`
	`>>>`

E6.4	Calcular o imposto de um valor de salário lido, considerando: se valor >3000: imposto = 25% senão se valor >2000: imposto = 20% senão se valor >1000: imposto = 10% senão : imposto = 5% Imprimir valor do salário, valor do imposto e salário liquido.

E6.5	Leia duas variáveis a, b. Leia o operador aritmético:{+,-,*, /}. Calcule o resultado da operação. Se o operador não existir exiba operador inválido.
	
	`a=float(input('a:'))`
	`b=float(input('b:'))`
	`oper=input('o:')`
	`if  oper=='+':val=a+b`
	`elif oper=='-':val=a-b`

```
elif oper=='*':val=a*b
elif oper=='/':val=a/b
elif oper=='^':val=a**b
else:
    print('operação invalida (+,-,*,/,^)')
    exit(1)
print('o valor de %5.2f %s %5.2f = %5.2f' % (a,oper,b,val))
>>>
a:3
b:7
o:+
o valor de  3.00 +  7.00 = 10.00
```

E6.6 Ler os minutos, min, da conta de celular. Se min for menor que 240 o preço é 0.20 de real; senão se min é menor que 120, o preço é 0.18 ao min; senão o preço é 0.15 ao min; imprima o valor da conta. Um só valor por minuto para toda a conta.

```
min=int(input('quantos min cel:'))
if min<120:
    preco=0.20
elif min<240:
    preco=0.18
else:
    preco = 0.15
print('pagar R$ %6.2f' % (preco*min))
```

E6.7 Ler o número de km percorridos e calcular a conta do Uber. Considere 90 centavos real por km, mas a partir dos 100 km o valor por km diminui para 80 centavos (para todo percurso).

Por exemplo, se a pessoa percorreu 90km. Vai pagar 90x0.90=81 reais. Mas se a pessoa andou 110km vai pagar 110x0.80=88 reais.

E6.8 Seja as cotações:

a) Dólar Turismo R$ 5,6000;
b) Euro R$ 6,20;
c) Libra R$ 6,9769
d) Peso Argentino R$ 0,0710;
e) Bitcoin 10,4200.

Isto é, para comprar 1 Dolar preciso pagar 5,6 reais e assim por diante.
Leia um caracter (a..e) e um valor em reais. Imprima o valor correspondente ao cambio, a compra na moeda estrangeira.

Se a pessoa digitar ‘a’ então 1000 reais. Então ela vai comprar 1000/5.6 dolares. Se a pessoa digitar ‘f’ então recebe a mensagem “opção invalida deve ser uma letra entre ‘a’ e ‘e’.

E6.9 Leia a,b,c três valores inteiros e imprima o intermediário. O intermediário não é o menor e nem o maior. Se os três são diferentes então temos menor, intermediário e maior. Podemos ter situações com valores iguais, então o intermediário pode ser igual a valores das pontas.

Por exemplo, se os três valores são 9,5,14; o intermediário é 9. Se os valores são 7,5,7 o intermediário é 7. Se os valores são 6,6,6 o intermediário é 6.

6.4.1 Perguntas conceituais (3)

	Responda com no mínimo 10 palavras e no máximo 20 palavras:
C6.1	Para que serve a combinação if/else?
C6.2	Para que serve a combinação if/elif ?
C6.3	Qual a diferença entre else e elif ?
C6.4	O que faz o exit(1) ?

Capítulo

7 REFATORANDO COM FUNÇÕES

Apresentamos o conceito de abstração, uso de funções definidas pelo programador, para organizar e refatorar o código dos programas. Por fim, apresentamos uma discussão do pensamento computacional sobre um kit mínimo de comandos necessários na programação.

7.1 Refatorando o código

Refatorar o código é reescrevê-lo de forma que ele ainda executa o mesmo comportamento, tenha o mesmo efeito, mas onde o código esteja escrito de uma forma mais simples, mais elegante, mais fácil de ler e compreender. No capítulo anterior, o programa que diz se é positivo, zero ou negativo foi reescrito de várias formas, qual é a melhor versão? É possível encontrar outra forma de escrita melhor?

Vamos fazer um programa que lê duas variáveis int e mostra o valor da maior delas (maior ou igual). Se forem iguais mostre uma delas.

```
#Ler dois valores e dizer qual é menor
x=int(input('digite x:'))
y=int(input('digite y:'))
print('valores lidos :',x, y)
if x<y:
     print('o menor é x:',x)
elif y<x:
     print('o menor é y:',y)
else:
     print('eles são iguais:',y)
>>>
digite x:6
digite y:3
valores lidos : 6 3
o menor é y: 3
```

Para dificultar mais, agora vamos fazer o mesmo com 3 valores. Ler os 3 valores e dizer quem é o menor.

```
#Ler 3 valores e dizer qual é menor
x=int(input('digite x:'))
y=int(input('digite y:'))
z=int(input('digite z:'))
print('valor lidos :',x, y,z)
if x<y and x<z:
     print('o menor é x:',x)
elif y<x and y<z:
     print('o menor é y:',y)
elif z<x and z<y:
     print('o menor é z:',z)
else:
     print('não tem só um mínimo:')
```

7.2 Refatorando com funções

Agora para refatorar estes códigos de escolha do menor, vamos introduzir o conceito de abstração. Uma função é uma abstração, dar um nome a um pedaço de código. Já introduzimos o conceito de função, já usamos algumas, mas como definir nossas próprias funções?

Uma função delimita um bloco de comandos, iniciando com a palavra def(ine) seguido de *nome*(), com ou sem parâmetros. No exemplo abaixo, em mini(a,b) entram as variáveis a,b (parâmetros) e sai ou a ou b, o menor dos valores. A saída é marcada por return (retorna).

```
def mini(a,b):
    if a<b: return a
    else  : return b
```

Para usar esta função apresentamos abaixo um programa que lê dois valores e chama a função mini() para escolher o menor. Def define a função que é chamada neste caso dentro de um comando de atribuição.

```
#ler dois valores e dizer quem é o menor
x=int(input('digite x:'))
y=int(input('digite y:'))
print('valores lidos :',x, y)
def mini(a,b):
    if a<b: return a
    else  : return b
m=mini(x,y) # chama a função
print('o menor é:',m)
```

Quanto a escrita do código podemos mudar a posição da função, podemos trazer a definição dela no início do código, antes da leitura dos valores, como segue.

```
#ler dois valores e dizer quem é o menor
def mini(a,b):
    if a<b: return a
    else  : return b
x=int(input('digite x:'))
y=int(input('digite y:'))
print('valores lidos :',x, y)
m=mini(x,y) # chama a função
print('o menor é:',m)
```

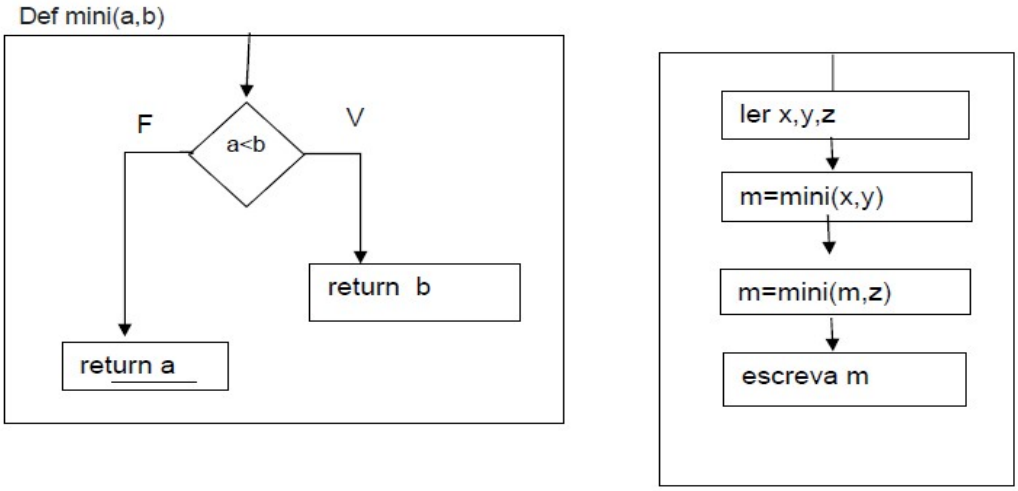
No cabeçalho da função entram os parâmetros, em mini(a,b) a e b são dois parâmetros **formais** (de forma). Em cada função deve existir um valor de saída que vem depois da palavra return. Tudo o que está indentado dentro da função é o seu corpo. Assim a função tem nome, parâmetros de entrada, um bloco de comandos com comandos de saída em seu corpo. Segue mais um exemplo, agora vamos ler 3 valores e dizer quem é o menor dos três.

```
#ler três valores e dizer quem é o menor
def mini(a,b):   # parametros formais
    if a<b: return a
    else  : return b
x=int(input('digite x:'))
y=int(input('digite y:'))
z=int(input('digite z:'))
print('valor lidos :',x, y, z)
m=mini(x,y)       #parametros efetivos
m=mini(m,z)
print('o(um) menor é:',m)
```

Utilizamos 2 linhas, uma para cada chamada, para definir quem é o valor mínimo, mas poderíamos ter usado uma versão do print mais complexa, com menos linhas: m=mini(mini(x,y),z). Aqui a função mini() está sendo reusada duas vezes, uma dentro da outra; os parâmetros nas chamadas de função são ditos **efetivos** (substituem os formais). Uma das grandes vitórias do pensamento computacional é o reuso de fragmentos de código, pois com isso os códigos ficam mais compactos, mais coesos e mais simples.

7.3 Fluxograma de função

Abaixo mostramos como fica o fluxograma de uma função, como uma caixa preta ele esconde o código da função mini() do algoritmo principal; no código principal a função é chamada num comando de atribuição. Neste exemplo a função mini é chamada duas vezes.



Fluxogramas da função mini() e do algoritmo escreve o menor de 3 valores

7.4 Pensamento computacional: refatorando com funções

No percorrer do texto, vimos vários conceitos de programação. Vamos nos aprofundar e familiarizar com eles aos poucos. Muitos assuntos são apenas introduzidos para mais adiante serem mais detalhados. Consideramos uma abordagem do geral para o específico, sendo que a cada capítulo novos conceitos são introduzidos e ou antigos conceitos são mais detalhados, por exemplo: apresentamos funções ainda no capítulo 2, agora aprofundamos um pouco mais, mas todo o capítulo 11 é sobre Funções.
Iniciamos com variáveis e expressões; depois vimos o comando de atribuição. Numa linha de comandos podemos encadear vários comandos de atribuição numa sequência (ex. a=1;b=3+a;x=1+b;print(x)), onde um é executado depois do outro em uma sequência, conforme a ordem de leitura. A sequência de comandos também pode seguir a ordem de leitura de cima para baixo. Chamadas de função também podem ser horizontais, sequenciadas em uma mesma linha, pois são parte de comandos de atribuição.
Para compor o kit de ferramentas de pensamento computacional, de pensamento de algoritmos, vamos precisar de mais alguns elementos. Na tabela abaixo apresentamos um kit mínimo, começando por variáveis, comandos de atribuição, comandos de sequência, comandos de entrada e saída, comandos condicionais, comandos de repetição e comandos de abstração.
Vamos nos aprofundar nos comandos de repetição nos próximos capítulos. Já apresentamos as funções como uma forma para organizar os códigos através do conceito de abstração, dar nome para um grupo de comandos.

Estruturados	Tipos de comandos	Exemplos
Verticais e/ou horizontais	1-Variáveis, objetos e expressões	a=1; c=[2,5,9]
	2-Comando de atribuição	
	3-Sequência de comandos	a=1;b=3+a;x=1+b;print(x)
	4-Comandos de entrada e saida	input() print()
Só verticais, só pela indentação	5-Comando condicional	if, elif, else:
	6-Comandos de repetição	for, while
	7-Mecanismo de abstração	def

Esta tabela apresenta os tipos de comandos necessários para codificação de qualquer algoritmo em Python. Os primeiros 4 tipos podem ser estruturados verticalmente ou horizontalmente, mas os 3 últimos só verticalmente pela indentação.

7.5 Exercícios (5)

Para cada exercício executar com pelo menos uma entrada de dados, para verificar se a resposta está correta.

E7.1	Faça a função maxi() que retorna o máximo de dois valores. Similar a função mini().
E7.2	Usando maxi() leia a,b,c,d encontre o máximo dos 4 valores.
E7.3	Usando maxi() leia a,b,c e escreva do maior para o menor. Por exemplo, a=7,b=15,c=2 a saída deve ser 15 7 2. Sugestão de pseudo-código: ler a,b,c max=maxi(a,maxi(b,c)) se a=max então: se b>c imprima (a, b, c) senão imprima (a, c, b) senão b=max... senão c=max...
E7.4	Faça a função inter(médio) de três valores. Por exemplo, a=7,b=15,c=2 o intermédio é 7. De a=15,b=15, c=7. O intermédio é 15. Etc. Sugestão de pseudo-código: Def inter(a,b,c): if a<=b<=c or c<=b<=a return b elif ...
E7.5	Usando maxi(), inter() e mini() leia a,b,c e escreva do maior para o menor. Por exemplo, a=7,b=15,c=2 a saída deve ser 15 7 2.

7.5.1 Perguntas conceituais (6)

	Responda com no mínimo 10 palavras e no máximo 20 palavras:
C7.1	O que é bloco de comandos?
C7.2	Qual é o papel da indentação?
C7.3	Quais são os tipos de comandos estruturados pela indentação?
C7.4	Quais os comandos que podem ser programados numa sequência horizontal dentro de uma mesma linha?
C7.5	O que é refatorar código?
C7.6	O que é reuso de código?
C7.7	O que é um pseudo código?
C7.8	O que é um fluxograma?

Capítulo 8 LISTA, TUPLA E FAIXA DE VALORES

Neste capítulo apresentamos listas e tuplas. São os objetos computacionais que estão associados a estruturas de dados para representação da repetição de elementos como vetores da matemática. Mostramos o comando chamado "for" para percorrer as listas. São introduzidos os conceitos de índices positivos, para caminhar para frente e índices negativos, para caminhar para trás.

8.1 Listas

lista=	**['a',**	**'b',**	**'c',**	**'d',**	**'e',**	**'f',**	**'g']**
índice	0	1	2	3	4	5	6
índice (-)	-7	-6	-5	-4	-3	-2	-1

Uma lista é um conjunto de valores indexados por um número inteiro chamado de índice, que inicia em zero. Os elementos contidos em uma lista podem ser de qualquer tipo, até mesmo outras listas, e não precisam ser todos do mesmo tipo. len() retorna o tamanho da lista.

```
>>>lista= ['a','b','c','d','e','f']
>>>len(lista)
7
>>>lista = [1, 2, 3]
>>>len(lista)
3
```

O índice permite acesso individual aos elementos, bem como permite trabalhar com as fatias. Com índices negativos caminhamos na lista de trás para frente, onde -1, -2 são o índice do último e penúltimo elementos.

```
>>> lista=[5,10,15]
>>> lista[0]
5
>>> lista[1]
10
>>> lista[-1]
15
```

8.2 Índices e fatiamento

Range(), faixa, intervalo é uma função iterável, que cria um intervalo. A sintaxe é range(stop) ou range([star],stop, step). Isto é, início, fim e passo. **Importante**: no range() o intevalo é sempre aberto no final. Se for até 10 na verdade o 10 é excluído, vai só até 9. list(range()) força o desenrolar do objeto iterável dentro de uma lista.

Faixa de valores, intervalo

range(início, fim, passo)
range(início, fim)
range(fim)

```
>>>range(10)
range(0, 10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
list(range(10,0,-1))
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> list(range(0,10,2))
[0, 2, 4, 6, 8]


>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(2,10))
[2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(2,10,2)) # só pares
[2, 4, 6, 8]
>>>list(range(1,10,2)) # só impares
[1, 3, 5, 7, 9]
```

Fatias são segmentos da lista. O segmento é acessado por uma faixa de valores. Numa lista podemos escrever: lista[início:fim]. Como em range(), do início até o fim, **exclusive** o fim. :2 denota, 0:2, então é 0 e o 1. Quando não diz o início é zero. Quando não diz o fim é até o último, ex. 1:, é tudo a partir do 1.

```
>>> lista=[5,10,15]
>>> lista[:2]
[5, 10]
>>> lista[1:]
[10, 15]

>>>lista= ['a','b','c','d','e','f','g']
>>>lista[1:4]
['b','c','d']
>>>lista[-6:-3]
['b','c','d']
```

É bem chato escrever os caracteres entre apóstrofos. Então podemos iniciar uma lista com as letras de 'a' até 'g' a partir de um string. Claro que podemos também converter uma lista de chars num string com a função "".join(). Seguem alguns exemplos.

```
>>> lista=list('abcdefg')
>>> lista
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> lista[1:4]
['b', 'c', 'd']

>>> str(lista[1:4])
"['b', 'c', 'd']"

>>> ''.join(lista[1:4])
'bcd'
>>> "".join(lista)
'abcdefg'
```

8.3 Acrescentar e remover elementos

Para modificar uma lista podemos atribuir valores para elementos indexados, podemos acrescentar elementos no fim ou podemos remover elementos. A função append() acrescenta um elemento no final da lista; a função remove() tira um item de uma lista.

append	anexe
remove	remove

```
>>> lista=[5,10,15]
>>> lista[2]=11
>>> lista.append(20)
>>> lista
[5, 10, 11, 20]
>>> lista.remove(10)
>>> lista
[5, 11, 20]
```

8.4 Comando for para percorrer listas

Para i em [2, 4, 88, 100]: escreva(i)

For i in [2, 4, 88, 100]: print(i)

Listas são facilmente percorridas com o comando for. A primeira forma é ver a lista como um conjunto e com o operador in (pertence) os elementos são acessados: elemento **in** lista. O end='' dentro do print() diz para colocar um branco depois de cada impressão.

```
>>>lista = [11, 22, 33]
>>>for i in lista: print(i, end=' ')
11 22 33
```

A segunda forma de acessar os elementos e a partir de seus índices, extraídos de uma faixa de valores, range(), que também é como uma lista de indices. Sempre use **len(lista)** para definir o fim da lista.

```
>>>lista = [11, 22, 33]
>>>for i in range(len(lista)): print(lista[i], end=' ')
11 22 33
```

Estes comandos de linha também podem ser agrupados num programa como ilustrado abaixo. O print() vazio envia o cursor para a próxima linha.

```
lista = [11, 22, 33]
for i in lista: print(i, end=',')
print()
for i in range(len(lista)): print(lista[i], end=' ')
>>>
11,22,33,
11 22 33
```

8.5 Tuplas

tupla=	**('a',**	**'b',**	**'c',**	**'d',**	**'e',**	**'f',**	**'g')**
índice	0	1	2	3	4	5	6
Índice (-)	-7	-6	-5	-4	-3	-2	-1

O tipo **tupla** é similar a uma lista, mas é **imutável, não podemos mudar o valor de seus elementos**. Sintaticamente, uma tupla é similar a uma lista de valores separados por vírgulas, mas a tupla é delimitada por parênteses. Na tupla, o acesso aos elementos e o fatiamento segue a mesma sintaxe da lista.

```
>>>tupla=     ('a','b','c', 'd','e','f','g')
>>>len(tupla)
7
>>>tupla[1:4]
('b','c','d')
>>>tupla[-6:-3]
('b','c','d')

>>>tupla = (10, 20, 30)
>>>len(tupla)
3

>>> tupla1=(20,40,60)
>>> for t in tupla1: print(t, end=' ')
20 40 60
```

Também o commando for permite percorrer uma tupla, usando a mesma notação que apresentamos para listas.

Pensamento computacional dos **tipos mutáveis e imutáveis**. Vimos 3 tipos compostos: strings, que são compostos de caracteres; listas e tuplas, que são compostas de elementos de qualquer tipo. Uma das diferenças entre strings e listas é que os elementos de uma lista

podem ser modificados, mas os caracteres em uma string não; tuplas também não podem ser modificadas. Em outras palavras, strings e tuplas são **imutáveis** e listas são **mutáveis**.

```
>>> tupla1=(20,40,60)
>>> tupla1[2]=0
TypeError: 'tuple' object does not support item assignment

>>> str='abc'
>>> str[1]='B'
TypeError: 'str' object does not support item assignment
```

8.6 Lista de exercícios (7)

Para cada exercício executar com pelo menos uma entrada de dados (se tem leitura) ou senão somente rodar, para verificar se a resposta está correta.

E8.1	Usando range() defina uma lista com os valores de 80 até 100.
	`>>> list(range(80,100))`
	`[80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]`

E8.2	Usando range() defina uma lista com os valores de 80 até 100, inclusive os extremos.
	`>>> list(range(80,101))`
	`[80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100]`

E8.3	Usando range() defina uma lista com os valores de 100 até 80, inclusive os extremos, em ordem decrescente.
E8.4	Usando range() defina uma lista com os valores de 80 até 100, só os valores pares.
E8.5	Usando range() defina uma lista com os valores de 80 até 100, só os valores impares. E usando um comando for imprima a lista gerada.

E8.6	Ler um número e imprimir todos os pares até o número lido. Use o comando for.
	`x=int(input('dig X:'))`
	`for i in range(0,x+1,2):`
	`  print(i, end=' ')`
	`print()`
	`>>>`

E8.7	Ler um número e imprimir todos os impares até o número lido. Use o comando for.

8.6.1 Perguntas conceituais (5)

	Responda com no mínimo 10 palavras e no máximo 20 palavras:
C8.1	Qual a diferença de lista para tupla?
C8.2	O que é uma range()?
C8.3	Qual a diferença entre uma fatia e uma faixa de valores?

C8.4	O que é um índice positivo?
C8.5	O que é um índice negativo?

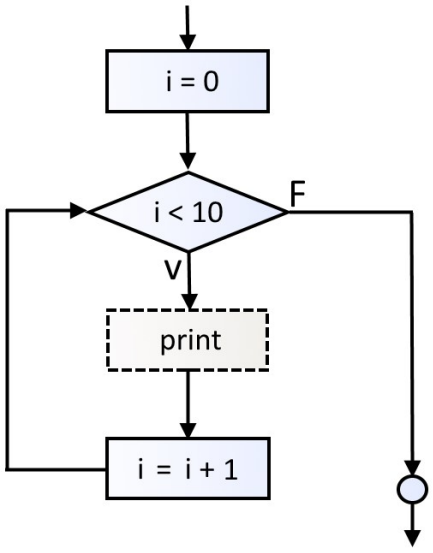
Capítulo

9 COMANDO DE REPETIÇÃO: FOR

Este capítulo aprofunda o estudo do comando for, que é um comando de repetição. Este comando foi introduzido no capítulo sobre listas. Este comando está fortemente associado a função range() e ao caminhamento sobre listas. Novos conceitos são apresentados: compreensão de listas; somar e contar com o for; comandos break e o continue. Vamos aprender a trabalhar como uma lista de valores digitados como uma string.

9.1 O comando FOR

Já introduzimos o comando for para percorrer listas e tuplas. Neste item vamos aprofundar o estudo do comando for. Ele permite percorrer qualquer objeto iterável, que simula o comportamento de uma faixa de valores. Além disso, vamos ver os comandos break e continue dentro do corpo do comando for.



Fluxograma do comando for

Acima temos a figura de um fluxograma do comando for. Ele executa seu bloco de contexto delimitado pela indentação; abaixo o primeiro print está dentro do for.

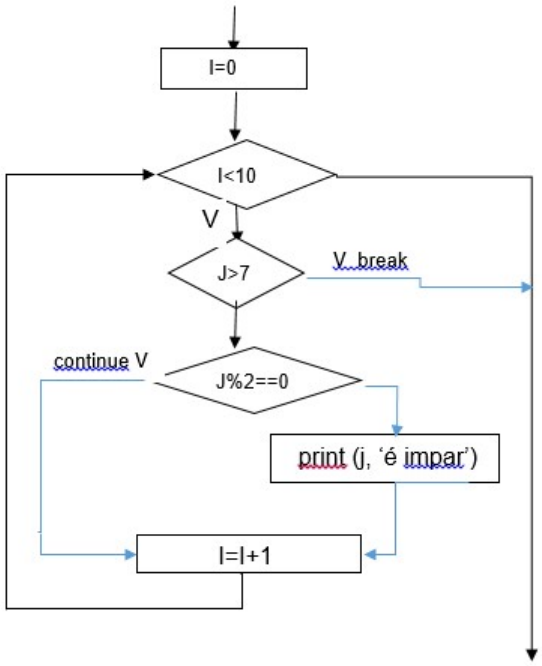
```
for i in range(0,10):
     print ("i=", i, end=' ')
print()
>>>
```

```
i= 0 i= 1 i= 2 i= 3 i= 4 i= 5 i= 6 i= 7 i= 8 i= 9
```

No comando for abaixo o laço é interrompido com break: quando o j==7 o break (parar) é ativado e o comando pára a sua execução. Aqui testamos se o j%2!=0 então o valor é impar. Note que só os impares são impressos.

```
for j in range(0,10):
  if j % 2!=0:  print(j, 'é impar')
  if j==7: break
>>>
1 é impar
3 é impar
5 é impar
7 é impar
```

Idem acima, mas com o novo comando continue. Se j%2==0, continue volta para o início do laço então não imprime 'é impar'. O break sai totalmente do laço, como mostra o fluxograma abaixo.



Fluxograma do Break e Continue

```
for j in range(0,10):
  if j>7: break
  if j % 2==0:continue
  print(j, 'é impar')
>>>
1 é impar
3 é impar
5 é impar
7 é impar
```

9.2 Somar com FOR

Somar todos os valores de 0 até 20. Vamos usar uma variável soma para armazenar o valor da soma, e vamos utilizar um range(0,21) pois o intervalo é aberto no final.

```
soma=0
for i in range(0,21):
  soma = soma+i
print(soma)
>>>
210
```

9.3 Compreensão de listas

Agora vamos aprender a trabalhar com uma string de dígitos, criada com a função split(). Aqui introduzimos outra notação chamada de **compreensão de lista**, que é baseada numa versão do comando for: y=[int(i) for i in x]. Todos os strings de x são transformados em int() em y.

```
>>> x= "12 17 21 2 47".split()
>>> x
['12', '17', '21', '2', '47']
>>> y=[int(i) for i in x]
>>> y
 [12, 17, 21, 2, 47]
```

A lista de string foi convertida numa lista de int. Int pode ser somado algebricamente. O split() pode também quebrar na vírgula. Quando usamos soma += i leia-se soma = soma+i. Este comando += evita de repetirmos o nome da variável soma, para acumular algo com o comando de atribuição.

```
>>> z="10, 60, 20, 50"
>>> z1=z.split(',')
>>> z1
['10', ' 60', ' 20', ' 50']
>>> z2=[int(i) for i in z1]
>>> z2
[10, 60, 20, 50]
>>>soma=0
>>>for i in z2: soma +=i
>>>soma
140
```

BOX: compreensão de lista é uma versão computacional da representação de conjuntos da matemática. Nela, podemos representar conjuntos por extensão ou por compreensão. Abaixo o conjunto A está representado por extensão e em B o mesmo conjunto está representado por compreensão.

A={ 1, 3, 5, 7, 9, 11, 13, 15 }

B={ x | x é impar, 0<x<16 }

Exemplo 1. Ler uma lista de valores int e somar os valores lidos. Neste caso vamos usar uma função split() para quebrar o string em valores. E cada valor em dígito deve ser convertido para numérico.

```
str1=input('digite valores (x,y,z,...):')
lis1=str1.split(',')
```

```
print(lis1)
lis2=[int(i) for i in lis1]
print(lis2)
soma=0
for i in lis2: soma +=i
print(soma)
>>>
digite valores (x,y,z,...):3,34,7,24,5
['3', '34', '7', '24', '5']
[3, 34, 7, 24, 5]
73
```

Exemplo 2. Ler uma lista do tipo "a2, f6, b20, c300, d5". Separar em duas listas uma das letras e outra dos valores, assim ['a', 'f', 'b', 'c', 'd'] [2, 6, 20, 300, 5]. O primero passo é quebrar na virgula.

```
lis= "a2, f6, b20, c300, d5"
#lis=input('digite valores:')
lis1=lis.split(',')
print(lis1)
>>>
['a2', ' f6', ' b20', ' c300', ' d5']
```

Note que na frente de alguns dos itens tem um caractere branco. Como ilimina-lo? com uma compressão de lista, com for para todos elementos da lista e com strip() que tira brancos noinício e no fim do string.

```
lis1=[i.strip() for i in lis1]
print(lis1)
>>>
['a2', 'f6', 'b20', 'c300', 'd5']
```

Agora fica fácil separar a lista em duas. Se X='c300' então x[0]= 'c' e x[1:]= '300' e int(x[1:])=300.

```
itens=[i[0] for i in lis1]
qti  =[int(i[1:]) for i in lis1]
print(itens,qti)
>>>
['a', 'f', 'b', 'c', 'd'] [2, 6, 20, 300, 5]
```

Exemplo 3. Idem ao exemplo 2, mas sem usar compreensão de listas. Vamos substituir a compreensão de listas por um comando for e com o método append para anexar os elementos na lista. A idéia é a partir de lis1, criar uma nova lista lis2 com os strings sem os brancos iniciais.

```
lis= "a2, f6, b20, c300, d5"
lis1=lis.split(',')
print(lis1)
lis2=[]
for i in lis1:
  lis2.append(i.strip())
print(lis2)
>>>
['a2', ' f6', ' b20', ' c300', ' d5']
```

```
['a2', 'f6', 'b20', 'c300', 'd5']
```

Agora podemos separar as letras dos valores em duas novas listas itens e qti. O comando for da compreensão de listas é convertido para um comando for onde no seu corpo os elementos da lista são anexados formando uma nova lista; sempre é necessário inicializar a lista antes do comando for.

```
itens=[] # inicializando a lista
for i in lis2:
    itens.append(i[0])
print(itens)
qti  = []
for i in lis2:
    qti.append(int(i[1:]))
print(qti)
>>>
['a', 'f', 'b', 'c', 'd']
[2, 6, 20, 300, 5]
```

Exemplo 4. Criar uma lista com comprensão de lista e range(). Vamos criar uma lista com os valores de 0 a 20.

```
>>> L=[i for i in range(0,21)]
>>> L
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, … 19, 20]
```

Agora só os múltiplos de 2 e outra com os múltiplos de 7.

```
>>> L=[i for i in range(0,21,2)]
>>> L
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

>>> L=[i for i in range(0,40,7)]
>>> L
[0, 7, 14, 21, 28, 35]
```

Sem compreensão de lista criar a lista dos pares entre 0 e 20.

```
L=[]
for i in range(0,21,2):
    L.append(i)
print(L)
>>>
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

9.4 Contar com FOR

Contar quantos valores de 0 até 20, são múltiplos de 3. Vamos usar uma variável conta para armazenar o valor da conta, e vamos utilizar x%3==0 para testar os divisores de três.

```
conta=0
for x in range(0,21):
  if x%3==0:
     conta = conta+1
print(conta)
>>>
```

Ler uma lista de valores int e contar quantos são impares. Mostrar a lista lida e a quantidade de impares. No exemplo abaixo, o '\n' denota o caractere quebra de linha.

```
str1=input('digite valores (x y z ...):')
lis1=str1.split()
print(lis1)
lis2=[int(i) for i in lis1]
conta=0
for i in lis2:
    if i % 2!=0: conta +=1
print(lis2, '\nNro de impares:', conta)
>>>
digite valores (x y z ...):2 5 5 7 4 4 2 3 23 20
['2', '5', '5', '7', '4', '4', '2', '3', '23', '20']
[2, 5, 5, 7, 4, 4, 2, 3, 23, 20]
Nro de impares: 5
```

9.5 Lista de exercícios (12)

Para cada exercício executar com pelo menos uma entrada de dados (se tem leitura) ou senão rodar para verificar se a resposta está correta.

E9.1	Ler dois valores a e b, e simular a multiplicação utilizando somente somas e subtrações. Por exemplo, 4x3=3+3+3+3 que resulta em 12.
	
	`a=int(input('a:'));b=int(input('b:'))`
	`mult=0`
	`print('o resultado de %d * %d=' % (a,b), end=' ' )`
	`for i in list(range(a)):`
	`  print('+',b, end=' ')`
	`  mult=mult+b`
	`print('é igual:', mult)`
	`>>>`
	`a:4`
	`b:7`
	`o resultado de 4 * 7= + 7 + 7 + 7 + 7 é igual: 28`

E9.2	Ler um número e calcular seu fatorial. Fatorial de 5 é 5*4*3*2*1, que 120. Use o comando for.
E9.3	Ler um número e imprimir todos os pares até o número lido. Use o comando for.
E9.4	Ler um número e imprimir a multiplicação de todos os pares de 2 até o número lido. Use o comando for. Se for lido 7, o valor é 2*4*6=48.
E9.5	Imprima a soma dos múltiplos de 7 entre 0 e 1000.
E9.6	Imprima a multiplicação dos múltiplos de 5 entre 1 e 1000.

E9.7	Imprima a seqüência de 9 até 0. Use o comando for.
	
	`for i in range(9,-1,-1):`
	`  print(i, end=' ')`

```
>>>
9 8 7 6 5 4 3 2 1 0
```

E9.8 Imprima a tabuada de um número n lido, como segue. Use o comando for e uma composição de string.

```
>>>
tabua do n:3
3 x 0 = 0
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
3 x 6 = 18
3 x 7 = 21
3 x 8 = 24
3 x 9 = 27
3 x 10 = 30
```

E9.9 Escreva todos valores de 51 até 100 com quebra linha de 10 em 10 valores. Use for.

```
for x in range(51,101):
    print(x, end=' ')
    if x % 10 ==0: print()
>>>
51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80
81 82 83 84 85 86 87 88 89 90
91 92 93 94 95 96 97 98 99 100
```

E9.10 Ler uma lista de supermercado com o produto, a quantidade e o tipo "a2, f6, b20, c2". Onde produtos são letras de a..f e os valores são desta tabela: {a:10, b:60, c:40, d:70, e:15, f:100}. Imprimir o boleto na seguinte forma:

```
"""
Item qty preço subtotal
a    2    10    20
f    6   100   600
b    20   60  1200
c    2    40    80

____________________
Total          1900 """

lis= "a2, f6, b20, c2"
#lis=input('digite valores:')
lis1=lis.split(',')
lis1=[i.strip() for i in lis1]
print(lis1)
itens=[i[0] for i in lis1]
qti  =[int(i[1:]) for i in lis1]
print(itens,qti)
def preco(x):
    '''{a:10, b:60, c:40, d:70, e:15, f:100}'''
    if   x=='a': p=10
    elif x=='b': p=60
```

```
    elif x=='c': p=40
    elif x=='d': p=70
    elif x=='e': p=15
    elif x=='f': p=100
    else: p=None
    return p
total = 0
print('item qti preço subtotal')

for i in range(len(itens)):
    subtotal = qti[i]*preco(itens[i])
    print(itens[i],qti[i],preco(itens[i]),subtotal)
    total += subtotal
print('total:', total)
>>>
['a2', 'f6', 'b20', 'c2']
['a', 'f', 'b', 'c'] [2, 6, 20, 2]
item qti preço subtotal
a   2  10  20
f   6  100 600
b   20 60  1200
c   2  40  80
total: 1900
```

Três apóstrofos ou três aspas neste programa, delimitam comentários que podem ser de múltiplas linhas.

9.5.1 Perguntas conceituais (3)

	Responda com no mínimo 10 palavras e no máximo 20 palavras:
C9.1	O que faz o comando break?
C9.2	O que faz o comando continue?
C9.3	O que é compreensão de lista?

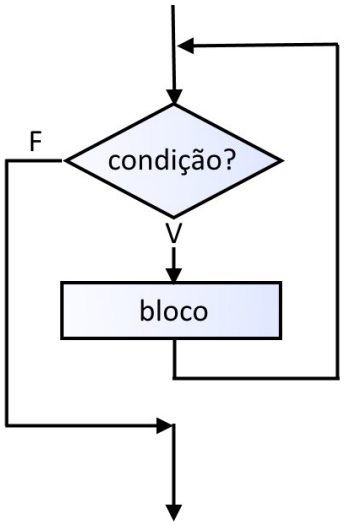
Capítulo

10 COMANDO DE REPETIÇÃO: WHILE

Este capítulo apresenta o comando while, que é um comando de repetição. Este comando é alternativo ao for, mas é até mais geral que ele. Vamos repetir muitos dos algoritmos que foram feitos com for, agora com o uso do comando while. Contar e somar com while. Percorrer estruturas e listas.

10.1 Comando WHILE

Abaixo temos o fluxograma do comando while: o bloco de comando é executado enquanto a condição for verdadeira. Quando a condição for falsa a execução do comando termina.



Fluxograma do comando while

Segue abaixo o **pseudo código** do comando while, aqui chamado de "**faça enquanto**". Diferente do comando for, o programador tem que inicializar a variável de controle do laço (i=0) e também lembrar de incrementa-la dentro da indentação do laço (i=i+1).

Pseudo código do comando "faça enquanto":

```
i=0
faça enquanto i<10:
    escreva ("i=", i, end=' ')
    i=i+1
escreva('\n')
>>>
i= 0 i= 1 i= 2 i= 3 i= 4 i= 5 i= 6 i= 7 i= 8 i= 9
\n
```

Segue abaixo o while definido no Python. Como no comando for, seu bloco de contexto é definido pela endentação. A principal diferença entre o comando for e o comando while é que o for automaticamente executa a inicialização e os incrementos da variável de controle. No while fica para o programador fazer a inicialização (i=0) e o incremento (i=i+1). Um erro comum é esquecer o incremento e então o algoritmo entra em looping (laço infinito).

```
i=0
while i<10:
   print ("i=", i, end=' ')
   i=i+1
print()
>>>
i= 0 i= 1 i= 2 i= 3 i= 4 i= 5 i= 6 i= 7 i= 8 i= 9
```

O segundo exemplo de while é um laço com True, isto é, “faça enquanto é verdade”; assim roda sempre, neste caso o break é necessário para terminar a repetição.

```
j=0
while True:
  if j % 2!=0:  print(j, 'é impar')
  if j==7: break
  j=j+1
>>>
1 é impar
3 é impar
5 é impar
7 é impar
```

10.2 Somar com WHILE

Somar todos os valores de 0 até 20. Vamos usar a variável soma para armazenar o valor da soma.

```
soma=0;i=0
while i <=20:
  soma = soma+i
  i=i+1
print(soma)
>>>
```

10.3 Contar com WHILE

Contar quantos valores de 0 até 20 são múltiplos de 3. Vamos usar uma variável conta para armazenar o valor da conta, e vamos utilizar i%3==0 para testar os divisores de três.

```
conta=0;i=0
while i<=20:
   if i%3==0:
      conta = conta+1
   i=i+1
print(conta)
>>>
7
```

Idem mas agora reescrevendo com o comando continue. A estrutura do algoritmo muda um pouco. Se i%3!=0, for diferente de 0, então não conta. O continue permite executar somente uma parte dos comandos do bloco. Note que aqui o incremento da variável de controle foi trazido para o início do bloco.

```
conta=0;i=0
while i<=20:
   i=i+1
   if i%3!=0: continue
   conta = conta+1
print(conta)
>>>
7
```

Por fim podemos dizer que o comando while é mais geral que o for. Mas com o while não se pode fazer comprenssão de lista. No mais, o comando for como controle de um bloco de comandos pode ser totalmente simulado pelo comando while. Tudo o que o for faz o while faz. O while faz mais, por exemplo, pode fazer um laço infinito, “while True:”.

10.4 Lista de Exercícios (11)

Para cada exercício executar com pelo menos uma entrada de dados (se tem leitura) ou somente rodar verificando se a resposta está correta.

E10.1	Ler dois valores a e b, e simular a multiplicação utilizando somente somas e subtrações. Por exemplo, 4x3=3+3+3+3 que resulta em 12.
	
	`a=int(input('a:'));b=int(input('b:'))`
	`mult=0`
	`print('o resultado de %d * %d=' % (a,b), end=' ' )`
	`while a>0:`
	`  print('+',b, end=' ')`
	`  mult=mult+b`
	`  a=a-1`
	`print('é igual:', mult)`
	`>>>`
	`a:4`
	`b:7`
	`o resultado de 4 * 7= + 7 + 7 + 7 + 7 é igual: 28`

E10.2	Temos dois operadores especiais de divisão // e %, divisão inteira e o resto da

	divisão. Como podemos programa-los utilizando soma e subtração. Por exemplo, 14//3 =4 e 14%3=2. Ler dois números inteiros positivos e simular os operadores // e %.
	
	`a=int(input('a:'));b=int(input('b:'))`
	`conta=0`
	`print('o resultado de %d // %d' % (a,b) )`
	`while a>=b:`
	`  conta = conta + 1`
	`  print('-',b, 'conta:', conta);`
	`  a=a-b`
	`print(' divisão inteira (//) :', conta)`
	`print(' resto da divisão (%): ', a)`
	`>>>`
	`a:13`
	`b:4`
	`o resultado de 13 // 4`
	`- 4 conta: 1`
	`- 4 conta: 2`
	`- 4 conta: 3`
	` divisão inteira (//) : 3`
	` resto da divisão (%):  1`

E10.3	Ler um número e imprimir todos os impares até o número lido. Use o comando while.
	
	`x=int(input('dig X:'))`
	`y=1`
	`while y<=x:`
	`  print(y, end=' ')`
	`  y=y+2`
	`print('fim')`
	`>>>`
	`dig X:9`
	`1 3 5 7 9`

E10.4	Ler um número e imprimir todos os pares até o número lido. Use o comando while.
E10.5	Ler um número e imprimir a multiplicação de todos os pares de 2 até o número lido. Use o comando while. Se for lido 7, o valor é 2*4*6=48.
E10.6	Imprimir a soma dos múltiplos de 7 entre 0 e 1000. Use o comando while.
E10.7	Ler um número e calcular seu fatorial. Fatorial de 5 é 5*4*3*2*1, que 120. Use o comando while.
E10.8	Imprima a multiplicação dos múltiplos de 5 entre 1 e 1000. Use o comando while.

E10.9	Imprima a seqüência de 9 até 0. Use o comando while.
	`>>>`
	`9 8 7 6 5 4 3 2 1 0`

E10.10	Imprima a tabuada de um número n lido. Use o comando while.
	`>>>`
	`tabua do n:3`
	`3 x 0 = 0`

	3 x 1 = 3 3 x 2 = 6 3 x 3 = 9 3 x 4 = 12 3 x 5 = 15 3 x 6 = 18 3 x 7 = 21 3 x 8 = 24 3 x 9 = 27 3 x 10 = 30
E10.11	Escreva todos valores de 51 até 100 com quebra linha de 10 em 10 valores. Use while.

10.4.1 Perguntas conceituais (1)

	Responda com no mínimo 10 palavras e no máximo 20 palavras:
C10.1	Qual a diferença entre o comando for e o comando while, tem alguma coisa que só pode ser feita com for ou só com while ?

Capítulo

11 FUNÇÕES

*Este capítulo aprofunda o conceito de funções, como objetos computacionais. Apresentamos parametros opcionais, *args, retornando mais de um item. Lista como vetor e funções com vetores.*

11.1 Min e max e intermediário

No capítulo sobre refatoração apresentamos algumas funções bem básicas para motivar o reuso delas em outras funções mais complexas. Como ordenar 3 valores lidos, usando mini(), maxi() e inter()?

```
def mini(a,b):
    if a<b: return a
    else  : return b
def maxi(a,b):
    if a>b: return a
    else  : return b
def inter(a,b,c): # intermediário
    if   a<=b<=c or c<=b<=a: return b
    elif b<=c<=a or a<=c<=b: return c
    elif b<=a<=c or c<=a<=b: return a
def ordena3(a,b,c) :
    min_=mini(a,mini(b,c))
    max_=maxi(a,maxi(b,c))
    inter_=inter(a,b,c)
    return (min_, inter_, max_)
print(ordena3(7,1,4))
>>>
(1, 4, 7)
```

A ordena3 retorna uma tupla. Como já conhecemos listas e tuplas podemos ajustar este código e ordenar 4 valores sem muita dificuldade. Segue uma versão alternativa do ordena3, agora com o uso de listas, com exemplos de remoção de elementos de listas. No ordena4 temos usos do *(), desempacotamento de parâmetros, ver próximo item. O ordena4 reusa completamente o ordena3l.

```
def ordena3l(a,b,c): # com uso de lista
    min_=mini(a,mini(b,c))
    max_=maxi(a,maxi(b,c))
    lista=[a,b,c]
    lista.remove(min_)
    lista.remove(max_)
    return (min_, lista[0], max_)
def ordena4(a,b,c,d) :
    min_=mini(mini(a,b),mini(c,d))
    lista=[a,b,c,d]
    lista.remove(min_)
    return(min_, *ordena3l(*lista))

from itertools import permutations
todos4 = permutations([7, 10, 2, 0])
for i in todos4: print(i, ordena4(*i))
>>>
(7, 10, 2, 0) (0, 2, 7, 10)
(7, 10, 0, 2) (0, 2, 7, 10)
(7, 2, 10, 0) (0, 2, 7, 10)
...
```

No teste do código usamos a permutação dos 4 elmentos, 4!, que é 4*3*2=24 testes para ver se o código esta correto. Este exemplo mostra como as funções podem ser encadeadas, encaixadas, reusadas, refatoradas, reescritas e testadas.

11.2 Empacotando e desempacotando args de funções: *(tuple)

Como já vimos, podemos inicializar uma lista de letras a partir de uma string, facilitando o tabalho de escrita da lista.

```
>>> list("abcd")
['a', 'b', 'c', 'd']
>>> li=list("abcd")
>>> li
['a', 'b', 'c', 'd']
```

De forma similar, o comando tuple cria uma tupla.

```
>>> tuple("abcd")
('a', 'b', 'c', 'd')
```

O operador * na frente de uma tupla (ou lista) de valores, **dentro de uma chamada de** função, remove a tupla. Ele **desempacota os argumentos** da função. Veja abaixo.

```
>>> a=tuple("abcd")
>>> a
('a', 'b', 'c', 'd')
>>> *a
SyntaxError: can't use starred expression here
>>> print(*a)
a b c d
>>> print(a)
('a', 'b', 'c', 'd')
```

O operador * na frente de uma tupla (ou lista) de valores**, dentro de uma definição de** função, ele **empacota os argumentos** da função. Segue um exemplo com a definição de uma função soma com ***args empacotando**, aceitando vários valores inteiros como parâmetros.

```
def soma(*li): # empacota na definição
   som=0;
   for i in li: som+=i
   return som
print(soma(*[1,2,3,4,5,6,6])) # desempacota na chamada
print(soma(3,4,4,7))
print(soma(3,4))
>>>
27
18
7
```

11.3 Pensamento computacional: lista versus vetor

Na computação um vetor (array) é uma estrutura de dados tipo uma lista mas com todos elementos do mesmo tipo de dados, chamada de estrutura de dados **homogênea**. Em Python listas podem ter elementos de diferentes tipos de dados, dados **heterogêneos**, por exemplo, ListaHetero=[1, 2.3, [1,2], 'a', "abc"]. Mas também elas podem representar vetores com dados homogêneos. Se o vetor nunca precisa ser modificado então pode ser representado numa tupla.

```
VetorInt = [2, 6, 7, 9, 12, 0, 3]
VetorFloat = [3.0, 4.0, 0.0, 11.0, 117.5]
ListaHetero=[1, 2.3, [1,2], 'a', "abc"]
print(VetorFloat[2:4])
print(ListaHetero[2])
>>>
[0.0, 11.0]
[1, 2]
```

11.4 Função soma()

Uma lista representa também um vetor. Os índices começam sempre no 0 e terminam com o comprimento (len) menos 1. Como somar os elementos de um vetor, várias versões.

```
V=[1,2,7,11,23,6]
# Defina a soma dos elementos do vetor use for, sem usar índices
def soma(v):
    soma=0;
    for i in v:soma = soma+i
    return soma
# Defina a soma dos elementos do vetor use for,
# acessando os elementos com índices
def soma1(v):
    soma=0;
    for i in range(len(v)):soma = soma+v[i]
    return soma
```

Podemos fazer vários aprimoramentos nos códigos, um deles é usar o operador +=. Assim soma=soma+v, fica soma +=v. Podemos também somar usando while.

```
# Defina a função soma dos elementos do vetor use while
def soma2(v):
    soma=0;i=0
    while i<len(v): soma += v[i];i+=1
    return soma
print('  soma:',soma(V))
print(' soma1:',soma1(V))
print(' soma2:',soma2(V))
>>>
 soma: 50
soma1: 50
soma2: 50
```

Por fim vamos mostrar mais uma versão da função soma, sem usar índice, usando pop(), que remove sempre o último elemento.

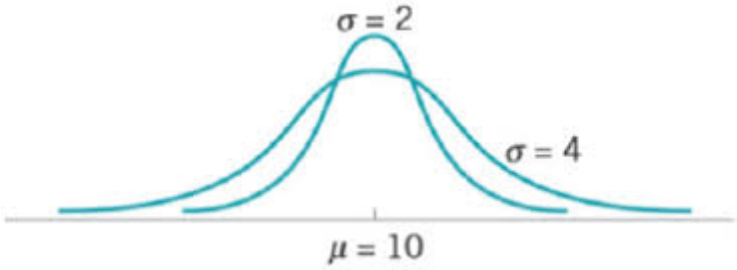
```
v=[1,5,7]
print('original :',v);
print('1ro pop  :',v.pop(), 'sobrou:',v)
print('2do pop  :',v.pop(), 'sobrou:',v)
>>>
original : [1, 5, 7]
1ro pop  : 7 sobrou: [1, 5]
2do pop  : 5 sobrou: [1]
```

Python tem uma versão simplificada para testar se uma lista é vazia na condição do while, por exemplo, “while v!=[]” pode ser escrito como “while v”; v é um vetor, na condição do while (ou if), quanto tem elementos é True, quando é vazio, [] é False.

```
>>> bool([])==False
True
>>> bool([1])==False
False

def soma2(v):
    soma=0;
    while v: soma += v.pop()
    return soma
V=[3,6,9,1,1,3,8]; print(' soma2:',soma2(V))
>>>
soma2: 31
```

11.4.1 Função média() e desvio()



Outra função bem usual é a média de um vetor de valores. A média é a soma dos elementos do vetor dividida pelo seu comprimento. Pode ser feita com sum() que é a função de soma interna do Python.

```
>>> sum([3,7,2,0,11])
23

#Defina a função média utilizando a função sum()
def med(v):return(sum(v)/len(v))

def med1(v):        # média sem utilizar sum()
    s=0;
    for i in range(len(v)): s=s+v[i]
    return s/len(v)
V=[1,2,7,11,23,6]
print('   med:',med(V))
print('  med1:',med1(V))
>>>
   med: 8.333333333333334
  med1: 8.333333333333334
```

Uma função parente da média é o desvio padrão. Seja SS o somatório da diferença de cada elemento com a média elevada ao quadrado. Assim SS = somatório de (V[i]-med)**2. O desvio é a raiz(SS/(len(V)). Um valor elevado a 0.5 é o mesmo que raiz quadrada, sqrt().

```
>>> import math
>>> math.sqrt(4)
2.0
>>> 4**0.5
2.0

def desvio(v):       # desvio padrão
    med=sum(v)/len(v);
    ss=0;
    for i in v: ss+=(i-med)**2
    return (ss/len(v))**0.5
V=[1,2,7,11,23,6]
print('  desv:',desvio(V))
>>>
  desv: 7.340905181848414
```

11.5 Retornando mais de um item, uma tupla

O método append() acrescenta um elemento no final de uma lista. Por exemplo,

```
>>> L=[]
>>> L.append(1)
>>> L
[1]
>>> L.append(5)
>>> L
[1, 5]
```

Agora vamos devolver dois objetos de uma função, dois vetores. Vamos primeiro mesclar dois vetores, da forma que se A=list("abcdefg") e B=list("1234567") então AB=list("a1b2c3d4e5f6g7"). Já a função separa faz o trabalho contrário, neste caso ela separa as duas listas, o A e o B. Note que o *separa(AB) remove a tupla, como argumento do print.

```
def mescla(A,B):  # entra A e B  e retorna AB
    AB=[]
    for i in range(len(A)):
        AB.append(A[i])
        AB.append(B[i])
    return ''.join(AB)

A=list("abcdefg")
B=list("1234567")
AB=list("a1b2c3d4e5f6g7")
print('mescla AB:', mescla(A,B))
print('mescla BA:', mescla(B,A))
>>>
mescla AB: a1b2c3d4e5f6g7
mescla BA: 1a2b3c4d5e6f7g

def separa(AB):     # retorna A e B
    A,B=[],[]
    for i in range(0,len(AB),2):
        A.append(AB[i])
        B.append(AB[i+1])
    return ''.join(A),''.join(B)
print('separa AB:', separa(AB))
print('separa BA:', separa(mescla(B,A)))
print()
print('separa AB:', *separa(AB))
print('separa BA:', *separa(mescla(B,A)))
>>>
separa AB: ('abcdefg', '1234567')
separa BA: ('1234567', 'abcdefg')

separa AB: abcdefg 1234567
separa BA: 1234567 abcdefg
```

11.6 *Mais sobre funções (* opcional)

Funções recebem opcionalmente parâmetros e devolvem opcionalmente resultados. A palavra def precede o nome da função. As funções também são consideradas objetos e têm alguns atributos, dentre os quais __doc__, o qual contém a documentação da função, e __name__, que contém o nome da função. Funções também podem ser atribuídas a variáveis. Funções são executáveis quando se coloca () no final, com ou sem parâmetros.
Também é possível utilizar parâmetros opcionais como por exemplo uma função que calcula a potência, onde poderíamos ou não informar o expoente. Caso não for informado é assumido o valor default 2.

```
def potência(base,exp=2):
  '''função potência com exp int opcional'''
  if exp==0: return 1
  pot=base
  for i in range(1,exp): pot=pot*base
  return pot

pot=potência
print(pot)           # imprime o objeto função
print(pot(2))        # 4
print(potência(2,8)) # 256
print(pot(4,3))      # 4*4*4
print(pot.__doc__)
print(pot.__name__)
>>>
<function potência at 0x0389FC88>
4
256
64
função potência com exp int opcional
potência
```

Para exemplificar fizemos pot=potência, aqui para a variável pot é atribuída a função potência. Note que não foi passado nenhum parâmetro entre ().
O poder das funções é abstrair fragmentos de código numa espécie de caixa preta. É dado um nome para uma sequência de comandos que executam uma funcionalidade.

11.7 Exercícios (10)

Ao responder estes exercícios, teste cada função com duas entradas para garantir que a saída está correta.

E11.1	Faça a função ordena5() que ordena os cinco parâmetros do menor para o maior
E11.2	Defina a soma dos elementos do vetor, use for, sem usar índices
E11.3	Defina a soma dos elementos do vetor, use for, acessando os elementos com índices
E11.4	Defina a função soma dos elementos do vetor use while, via índices
E11.5	Defina a função média utilizando a função soma

E11.6	Defina a função média, med1, sem utilizar a função soma (sum)
E11.7	Defina a função desvio padrão

E11.8	Faça a função mescla(A,B) que entra A e B e retorna AB
	`def mescla(A,B):`
	`  AB=[]`
	`  for i in range(len(A)):`
	`  AB.append(A[i])`
	`  AB.append(B[i])`
	`  return ''.join(AB)`
	
	`A=list("abcdefg")`
	`B=list("1234567")`
	`print('mescla AB:', mescla(A,B))`
	`print('mescla BA:', mescla(B,A))`

E11.9	Faça a função separa(AB) que retorna A e B
	`AB=list("a1b2c3d4e5f6g7")`
	`def separa(AB):`
	`  A,B=[],[]`
	`  for i in range(0,len(AB),2):`
	`  A.append(AB[i])`
	`  B.append(AB[i+1])`
	`  return ''.join(A),''.join(B)`
	`print('separa AB:', *separa(AB))`

11.7.1 Perguntas conceituais (4)

	Responda com no mínimo 10 palavras e no máximo 20 palavras:
C11.1	O que é uma estrutura de dados chamada de vetor?
C11.2	O que faz o operador *args?
C11.3	Cite dois atributos de uma função?
C11.4	O que é um parâmetro opcional?

Capítulo
12 FUNÇÕES RECURSIVAS

Apresentamos problemas e funções recursivas. Como extra, da cultura do pensamento computacional, falamos sobre paradigmas de programação e jeito pythônico de codificação.

12.1 Função reverse() e palíndromo()

A função reverse, reverter a lista, tem várias aplicações. Uma delas é no código da função palíndromo. Uma lista é um palíndromo se ela for igual a sua imagem invertida, assim list("abcba") é palíndromo, pois list("abcba") =rev(list("abcba")).

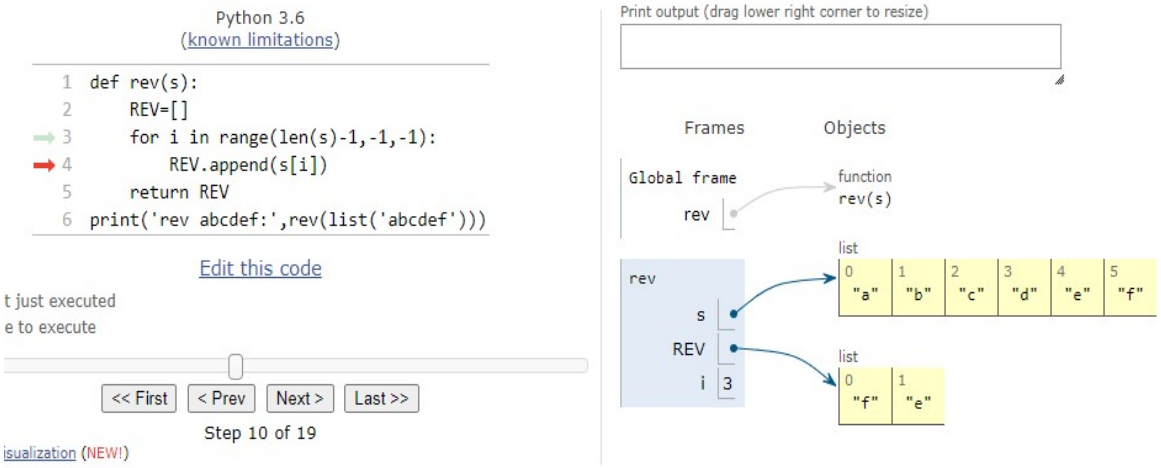
lista=		**['a',**	**'b',**	**'c',**	**'b',**	**'a']**
índice	-1	0	1	2	3	4

Uma forma simples de inverter uma lista é pela manipulação de fatias, assim reverso(x) = x[::-1]. Mas vamos programar outras formas para exercitar a solução do problema:

i) rev: iniciar no índice len()-1, até 0, diminuindo de 1; fica range(len(s)-1,-1,-1);
ii) rev1: percorrer a lista com for tirando do inicio da lista e colocando no final;
iii)rev2: idem com recursividade, sem o for.

```
def rev(s):
    REV=[]
    for i in range(len(s)-1,-1,-1):
        REV.append(s[i])
    return REV
print('rev abcdef:',*rev(list('abcdef')))
>>>
rev abcdef: f e d c b a
```

Abaixo com o Python tutor vemos a lista REV sendo construída na forma reversa. Passo a passo os elementos de trás -1 para frente (com passo -1) são copiados para a nova lista, REV.



O append sempre acrescenta um elemento no final da lista. Mas o + é mais flexível podendo concatenar quaisquer duas listas, assim podemos percorrer a lista do início para o fim e sempre ir acrescentando na frente da nova lista REV sendo formada. O + concatena duas lista portanto o elemento s[i] é colocado dentro de uma lista [s[i]].

```
def rev1(s):
    REV=[]
    for i in range(len(s)):REV=[s[i]]+REV
    return REV
```

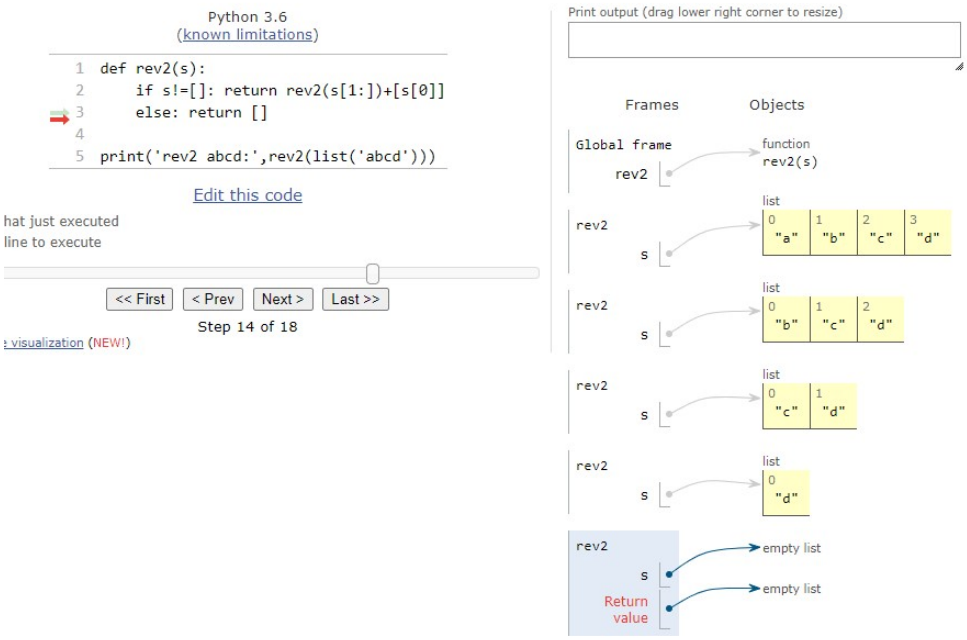
12.2 Funções recursivas

Outro recurso poderoso de programação são as funções recursivas. Vamos apresentar o conceito com a função reverse.

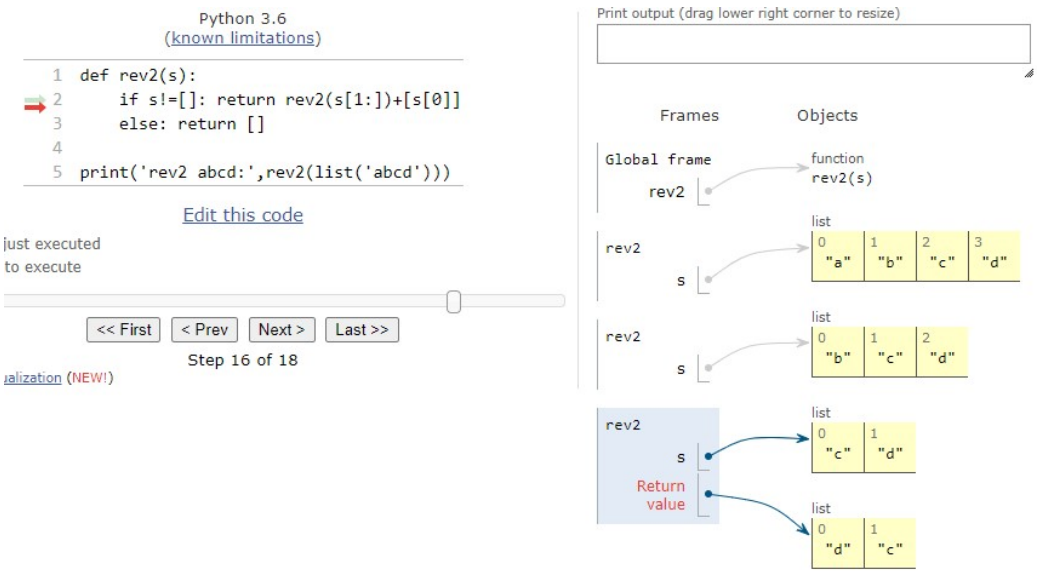
```
def rev2(s):
    if s!=[]: return rev2(s[1:])+[s[0]] # recursividade
    else: return []                     # base: fim
```

Uma **função é recursiva** se ela se chama a si mesma: na linha do def, ela é defindа, e dentro do seu corpo, ela já é chamada antes de termos terminado sua definição. Dentro do corpo: uma linha ela se chama recursivamente (caso recursivo) e na outra linha ela termina a recusividade (caso base).

Abaixo no Python Tutor vemos que primeiro a função vai se chamando e a lista de entrada diminui até ficar vazia. As funções recursivas vão se empilhando dentro da máquina abstrata de execução do Python. Depois de chegar no vazio da lista da entrada, as funções vão gerando o retorno, com as listas parciais invertidas.



Com mais dois passos next, vemos o retorno da lista parcial [c,d] invertida, agora [d,c]. Continuando com next as funções são removidas da pilha de execução e o resultado vem sendo montado, até se chegar em [d,c,b,a]. Para animar funções recursivas no Python Tutor utilize entradas pequenas de no máximo 5 elementos.



Abaixo temos um teste comparando as três versões de reverse, para mostar que elas geram o mesmo resultado.

```
li=list("123456")
print('rev:', *rev(li), 'rev1:', *rev1(li),'rev2:', *rev2(li)
>>>
rev: 6 5 4 3 2 1 rev1: 6 5 4 3 2 1 rev2: 6 5 4 3 2 1
```

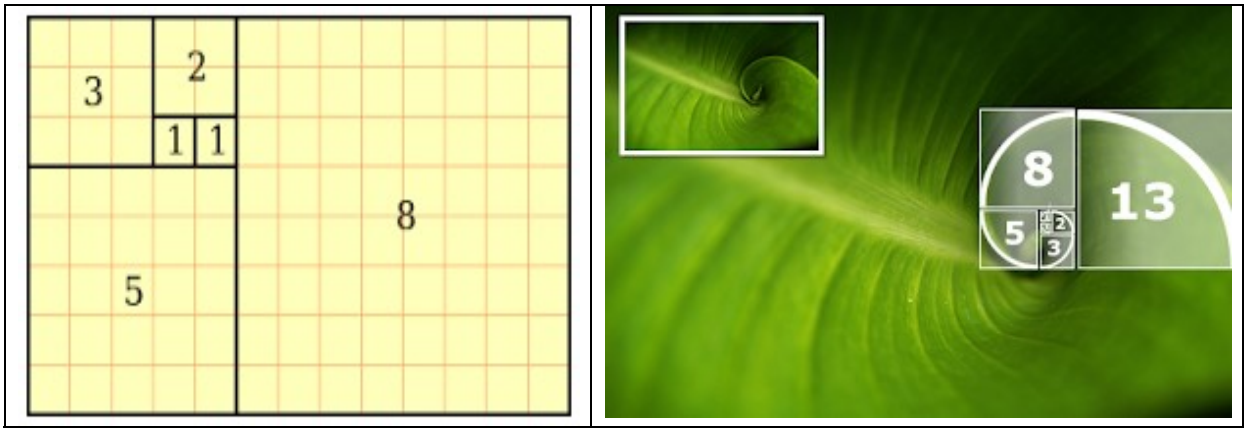
Agora que temos várias versões de reverse vamos programar duas versões da função palíndromo, a primeira mais fácil de ler, x==rev(x) e a segunda um pouco mais abstrata x==x[::-1].

```
def palídromo(x): return x==rev(x)
def palídromo1(x):return x==x[::-1]

print(palídromo(list('abcdeedcba')))
print(palídromo(list('12321')))
print(palídromo1(list('12321')))
print(palídromo1(list('123X21')))
>>>
True
True
True
False
```

12.3 Problemas recursivos

O que faz uma função recursiva? Porque utilizar funções recursivas? No exemplo de reverse, a recursividade substitui o comando de repetição. Mas, na matemática é comum o uso de funções recursivas para tratar de problemas com definições recursivas. Neste caso, a recursividade da linguagem de programação permite uma codificação direta da função matemática com a função a ser programada.



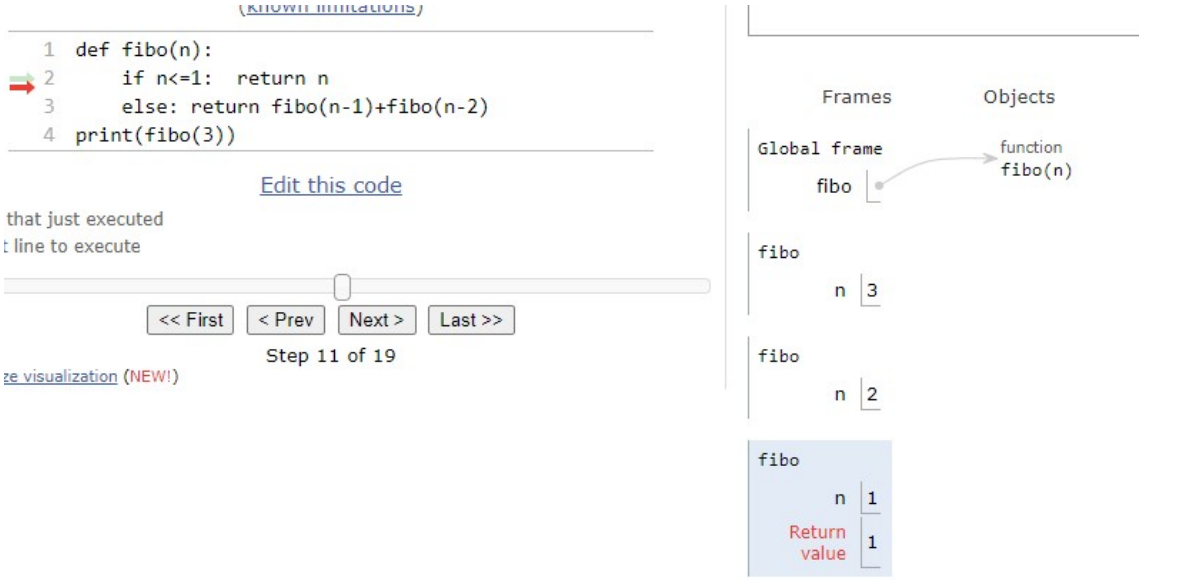
As folhas da Bromélia são formadas por espirais, cujos raios pertencem a série de Fibonacci. Esta série foi descoberta pelo matemático Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... Ele mostrou a série com o exemplo do crescimento acelerado da população de coelhos, com a fórmula recursiva:

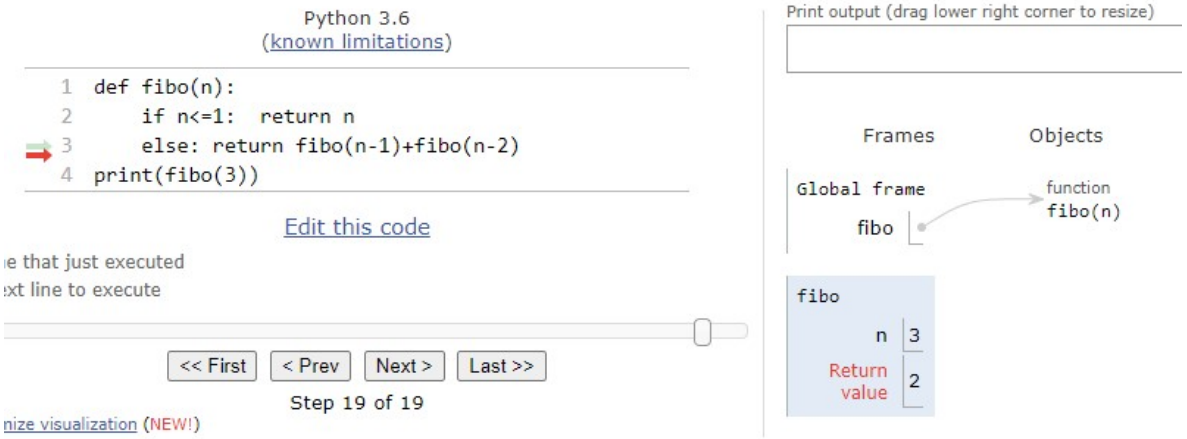
$F_0 = 0, F_1 = 1$ **# Caso base**
$F_n = F_{n-1} + F_{n-2}$ **# Caso recursivo**

Segue o programa recursivo que codifica esta série. São dois casos na fórmulas, então temos um comando if para decidir qual dos casos será executado a partir do valor de n; caso base ou caso recursivo. No teste abaixo com n=3, o terceiro elemento da série é 2 e o sexto elemento é o 8.

```
def fibo(n):
    if n<=1:  return n                # base: fim
    else: return fibo(n-1)+fibo(n-2) # recursividade
print(fibo(3))
print(fibo(6))
for i in range(0,10): print(fibo(i),end=' ')
>>>
2
8
0 1 1 2 3 5 8 13 21 34
```

Abaixo com o Python tutor podemos ver as várias chamadas de fibo() até termos um primeiro retorno para fibo(1). Com next e prev podemos navegar sobre a execução das chamadas recursivas, para compreender melhor como ocorre a recursão.





12.4 Exercícios (7)

Ao responder estes exercícios, teste cada função com duas entradas para garantir que a saída está correta.

E12.1	Faça a função reverse usando um for e range: iniciar no índice len()-1, até 0, diminuindo de 1; fica range(len(s)-1,-1,-1)
	`def rev(s):`
	`  REV=[]`
	`  for i in range(len(s)-1,-1,-1):`
	`  REV.append(s[i])`
	`  return REV`
	`print('rev abcdef:',rev(list('abcdef')))`

E12.2	Faça a função reverse: percorrer a lista do início para o fim com for e sempre ir acrescentando na frente da nova lista REV sendo formada, use o + para concatenar na frente de REV.
E12.3	Faça a função reverse com a estratégia da questão anterior, substituindo o for por **recursividade**: percorrer a lista do início para o fim e sempre ir acrescentando na frente da nova lista REV sendo formada, use o + para concatenar

E12.4	Faça a função palídromo usando x[::-1] para inverter x
	`def palídromo1(x):return x==x[::-1]`
	`print(palídromo1(list('12321')))`
	`print(palídromo1(list('123X21')))`

E12.5	Faça a função palídromo usando rev()
E12.6	Faça a função palídromo sem usar uma lista invertida. Comparando sempre os dois extremos, em direção ao meio. Use índices para fazer as comparações.
E12.7	Faça a função fatorial usando recursividade: i) Caso base: fatorial(n) se n<=1 é 1 ii) Caso recursivo: fatorial(n) se n>1 é igual a fatorial(n-1)*n

12.4.1 Perguntas conceituais (2)

	Responda com no mínimo 10 palavras e no máximo 20 palavras:
C12.1	O que caracteriza uma função recursiva?
C12.2	Numa função recursiva, qual a diferença entre o caso base e o caso recursivo?

12.5 Paradigmas de programação(nerd)

O uso da notação declarativa compreensão de lista aproxima a linguagem Python ao **paradigma de codificação funcional**, que contrasta com o paradigma procedural e com o paradigma lógico. O **paradigma procedural** é bem representado nos algoritmos que vimos até o momento, com excessão de alguns. A versão recursiva do reverso da lista tem estilo funcional. O programa Fibonacci tem estilo funcional. Por outro lado, o paradigma lógico é totalmente baseado em expressões lógicas. Vamos exemplificar o paradigma funcional e lógico com a função Fibonacci.

```
# em Python – funcional
def fibof(n): return n if n<=1 else fibof(n-1) + fibof(n-2)

""" em Prolog – lógico
% fib(in,out)
    fib(N,F) :- N<=1, F=N.
    fib(N,F) :- N >1, fib(N-1,F1), fib(N-2,F2), F is F1+F2.
"""
```

Neste código Prolog, do paradigma lógico, fib(In,Out) indica que entra N e sai F. Na regra fib(N,F) :- N<=1,F=N se lê: fibonacci de N é F se N<=1 e F=N. De forma similar se lê a outra regra para N>1. O código Prolog é equivalente a versão funcional, mas o retorno do valor é no parâmetro F. Os retornos parciais são em F1 e F2.

Em Python pode-se simular o estilo lógico de codificação. Vamos codificar uma função que retorna Bool mas que faz o papel de uma atribuição (=) chamada attr, que sempre retorna True; esta função pode fazer uma atribuição no meio de uma expressão lógica, no lugar de um operando. As variáveis N1,N2 e F são necessárias para pegar o parâmetro que retorna das chamadas das funções Bool. Em Python sómente estruturas de dados mutáveis podem retornar, via parâmetros, valores de funções; por isso, N, N1 e N2 são listas. Usa-se o primeiro elemento da lista com o retorno da função fiboL().

```
def attr(N,x):  # bool
    N[0]=x; return True
def fiboL(n,F): # bool
    N1=[0];N2=[0] # pars Out
    return attr(F,n) if n<=1 else fiboL(n-1, N1) and fiboL(n-2,N2) and attr(F,N1[0]+N2[0])
#
for i in range(0,10): # estilo procedural
    F=[0];fiboL(i,F);
    print(F[0],end=' ')
>>> 0 1 1 2 3 5 8 13 21 34
```

No paradigma funcional todos os parâmetros só entram, os valores de retorno devem retornar encima do nome da função. “Tudo” são funções. Por isso se chama paradigma funcional.

No código lógico acima, as duas funções são do tipo Bool, lógico. O corpo da função fiboL() é uma expressão lógica. “Tudo” são expressões lógicas. Por isso se chama paradigma lógico.

12.6 Box: Jeito pythônico, pythonic

Existem gírias de codificação em Python. Por exemplo, x[-1] é a gíria pythônica de pegar o último elemento do vetor x. Junto com isso x[::-1] é o jeito pythônico de inverter x.

Tim Peters criou 20 aforismos conhecidos como The Zen of Python. Seguem alguns:
Bonito é melhor que feio.
Explícito é melhor que implícito.
Simples é melhor que complexo.
Complexo é melhor que complicado.
Plano é melhor que aninhado.
Esparso é melhor que denso.
Legibilidade conta.
...

Para entender um pouco de alguns destes aforismos vamos ver três exemplos, de códigos em Python comparando com outras linguagens.

12.6.1 Exemplos de código pythônico

EX1: Permutar os valores de duas variáveis
Outras linguagens:

```
int x=1, y=2,temp;
temp=x; x=y; y=temp;
```

Em Python:

```
x=1; y=2
x,y = y,x
```

EX2: Operadores relacionais encadeados
Outras linguagens:

```
int x=1, y=2, z=3;
if (x<y and y<=z) { }
```

Em Python:

```
x=1; y=2; z=3;
if x<y<=z:
```

EX3: comando switch
Outras linguagens:

```
int num = 3;
switch (num) {
    case 1: função_caso1();
    case 2: função_caso2();
    default:...}
```

O modo pythônico de fazer é usando um dicionário:

```
caso = {
    1: função_caso1,
    2: função_caso2,
    3: ...}
```

E chamar a função assim: `caso[num]()`

EX3: percorrendo uma lista com chave/valor:

Outras linguagens:

```
int seq[5] = {7, 11, 23, 3, 4};
for (int i = 0; i < 5; i++) {
    printf("%d, %d\n", i, seq[i]);
}
```

Em Python:

```
seq = [7, 11, 23, 3, 4]
for key, value in enumerate(seq):
    print(key, value)
```

Capítulo

13 MÉTODOS DE BUSCA

Mostramos alguns métodos de busca em listas, como uma forma de exercitar o desenvolvimento de algoritmos e as técnicas de programação. Estes métodos são utilizados no próximo capítulo que trata sobre métodos de ordenação.

13.1 Função maxi() e imax()

Sobre listas, outras funções muito usuais são o máximo e o mínimo. Seguem as funções da linguagem Python.

```
>>> L=[7, 11, 23, 3, 4]
#      0   1   2  3  4  # indices
>>> max(L)
23
>>> min(L)
3
>>> L.index(max(L))
2
>>> L.index(min(L))
3
```

Podemos fazer, para efeito didático, a nossa versão destas mesmas funções. Vamos começar com três versões relacionadas com a função máximo:

i) Uma função maxi() que seja equivalente a max() do Python;

ii) Uma função imax() que retorna o índice do max() equivalente a L.index(max(L);

iii) Idem imax() mas que faça a busca sobre o vetor a partir de um índice dado como parâmetro, que é útil para fazer um método de ordenação.

```
#Defina a função que retorna o máximo elemento de um vetor
def maxi(v):
    M=v[0];
    for i in range(1,len(v)):
        if M<v[i]:M=v[i]
    return M
```

```
#Defina a função que retorna o índice do máximo elemento de um vetor
```

```
def imax(v):
    M=0;
    for i in range(1,len(v)):
        if v[M]<v[i]:M=i
    return M
print(' maxi:', maxi(V))
print(' imax:', imax(V))

# idem iniciando a busca em j
def imax1(j,v):
    M=j;
    for i in range(j,len(v)):
        if v[M]<v[i]:M=i
    return M
print(' maxi:', maxi(V))
print(' imax:', imax(V))
print(' imax:', imax1(3,V))
```

13.2 Função busca(), in1() e index1()

Python tem funções de busca em listas, in e index().

```
>>> V=[1,3,7,11,23,6]
>>> 3 in V
True
>>> 12 in V
False
>>> V.index(3)
1
>>> V.index(12)
Traceback (most recent call last):
…
ValueError: 12 is not in list
```

Note que index resulta em erro quando o valor buscado não está na lista. Vamos fazer nossas versões destas funções.

```
def in1(x,v):
    for i in v:
        if x==i:return True
    return False
def index1(x,v):
    for i in range(len(v)):
        if x==v[i]:return i
    return None

V=[1,3,7,11,23,6]
print('in1    3:',in1(3,V))
print('index1 23:',index1(23,V))
>>>
in1    3: True
index1 23: 4

V="aafdskjg  kdnfgjkdazgjk"
```

```
print('in1    k:',in1('k',V))
print('index1 k:',index1('k',V))
>>>
in1    k: True
index1 k: 5
```

Note que testamos também as funções sobre uma string V. Foram feitas pensando em listas mas poder ser utilizadas em string e também em tuplas.

13.3 Busca binária

As buscas codificadas acima são lineares, elas caminham sobre todo o vetor. É possível fazer buscas mais inteligentes se o vetor estiver ordenado. Um método mais inteligente é busca binária, sobre um vetor ordenado. Podemos usar a função sorted() para ordenar um vetor.

```
>>> V=[1,3,7,11,23,6,3,1,24]
>>> V
[1, 3, 7, 11, 23, 6, 3, 1, 24]
>>> sorted(V)
[1, 1, 3, 3, 6, 7, 11, 23, 24]
```

Esta busca elimina metade do vetor a cada passo. Supondo que buscamos x=23 no vetor V, onde imin=0, imax=8, imed=4. No primero passo o imin = 4+1. Metade inferior do vetor foi cortada na busca. Sobrou V[5:]= [7, 11, 23, 24]. Este processo de sempre cortar a metade termina com o elemento encontrado ou quando imin>imax. Neste último caso, o x não esta no vetor.

```
 V     [1, 1, 3, 3, 6, 7, 11, 23, 24]
 Indice 0  1  2  3  4  5  6   7   8

1ro passo busca 23  em: [1, 1, 3, 3, 6, 7, 11, 23, 24]
2do passo busca 23  em: [7, 11, 23, 24]
3ro passo busca 23  em: [23, 24]
```

Esta busca trabalha considerando que o vetor é ordenado. O processo de busca compara o índice do meio (imed) com o elemento x buscado. O imed é igual a (imax+imin)//2. Assim quando comparamos x como o elemento V[imed] tem três situações:
Se x==V[meio] x foi encontrado, retorna o índice meio
Senão_se x<V[meio] x esta entre o valor de índice imin e meio-1;
Senão x>V[meio] x está entre o valor de índice meio+1 e imax.

```
V=[1,3,7,11,23,6,3,1,24]
V1=sorted(V)
def busca_bin(V, x):
    imin = 0          #(index)min
    imax = len(V)-1
    while imin <= imax:
        #print('busca', x, ' em:', V[imin:imax+1])
        imed = (imin+imax)//2
        if V[imed] == x: return imed
        elif x < V[imed]: imax = imed-1
        else: imin = imed+1
    return -1 # não esta no vetor
print(V1)
print(busca_bin(V1,23))
print(busca_bin(V1, 6))
print(busca_bin(V1,10))
>>>
[1, 1, 3, 3, 6, 7, 11, 23, 24]
7
4
-1
```

Uma busca linear em média percorre 50% do vetor para encontar um valor. Se o valor não existe no vetor é o pior caso ela percorre todo o vetor, testando o valor procurado contra todos elementos do vetor. Neste caso dizemos que a complexidade desse algoritmo é da ordem de O(n) onde n é o tamanho do vetor.

Já na pesquisa binária a complexidade do algoritmo e da ordem de $O(\log_2 n)$ para o pior caso. Assim por exemplo, num vetor de 100 elementos, n=100. No pior caso a pesquisa linear testa todos os 100. A binária no vetor de 100 testa em média 6.64 elementos pois $\log_2 100$ é igual a 6.64; se aumentamos para n=1000, o valor de $\log_2 1000$ é 9.96. Então a pesquisa binária é muito mais eficiente, muito mais rápida, mas só funciona em vetores ordenados.

13.4 Exercícios (9)

Ao responder estes exercícios, teste cada função com uma entrada para garantir que a saída está correta.

E13.1	Defina a função que retorna o máximo elemento de um vetor, use for
E13.2	Defina a função que retorna o índice do máximo elemento de um vetor

E13.3	Retorne o índice do máximo do vetor, iniciando a busca em j
	`def imax1(j,v):`
	`  M=j;`
	`  for i in range(j,len(v)):`
	`  if v[M]<v[i]:M=i`
	`  return M`
	`V=[3,6,9,1,1,3,8]; print(' imax:', imax1(3,V))`

E13.4	Defina mini() que retorna o mínimo elemento de um vetor
E13.5	Defina imin() que retorna o índice do mínimo elemento de um vetor

E13.6	Defina uma função busca similar ao in, in1(), use o comando for
	`def in1(x,v):`
	`  for i in v:`
	`  if x==i:return True`
	`  return False`

E13.7	Defina uma função similar ao index() que retorna o índice do elemento no vetor
	`def index1(x,v):`
	`  for i in range(len(v)):`
	`  if x==v[i]:return i`
	`  return None`
	`V=[1,3,7,11,23,6]`
	`print('index1 23:',index1(23,V))`

E13.8	Defina a função in1() usando um while
E13.9	Defina a função index1() usando um while

13.4.1 Perguntas conceituais (1)

	Responda com no mínimo 10 palavras e no máximo 20 palavras:
C14.1	Qual a diferença entre uma busca linear e uma busca binária?