



INTRODUÇÃO À PYTHON

Geraldo Xexéo

8 de Março de 2016

[COMPUTER SCIENCE] IS NOT REALLY ABOUT COMPUTERS – AND IT'S NOT ABOUT COMPUTERS IN THE SAME SENSE THAT PHYSICS IS NOT REALLY ABOUT PARTICLE ACCELERATORS, AND BIOLOGY IS NOT ABOUT MICROSCOPES AND PETRI DISHES...AND GEOMETRY ISN'T REALLY ABOUT USING SURVEYING INSTRUMENTS. NOW THE REASON THAT WE THINK COMPUTER SCIENCE IS ABOUT COMPUTERS IS PRETTY MUCH THE SAME REASON THAT THE EGYPTIANS THOUGHT GEOMETRY WAS ABOUT SURVEYING INSTRUMENTS: WHEN SOME FIELD IS JUST GETTING STARTED AND YOU DON'T REALLY UNDERSTAND IT VERY WELL, IT'S VERY EASY TO CONFUSE THE ESSENCE OF WHAT YOU'RE DOING WITH THE TOOLS THAT YOU USE.

HAL ABELSON

GERALDO XEXÉO

INTRODUÇÃO À PYTHON

AINDA NÃO PUBLICADO PELO AUTOR

Copyright © 2016 Geraldo Xexéo

AINDA NÃO PUBLICADO PELO AUTOR

WWW.XEXEO.NET

Este documento não está licenciado para cópia, armazenamento ou empréstimo.

Antes da Primeira Impressão, março 2016

Conteúdo

I	<i>Conceitos Iniciais</i>	15
1	<i>Introdução</i>	17
2	<i>Usando o Interpretador Python</i>	25
3	<i>Python como Calculadora</i>	29
4	<i>Tipos de Dados</i>	31
5	<i>Variáveis e Atribuição</i>	47
6	<i>Introdução a Funções</i>	51
7	<i>Usando Funções e Módulos</i>	53
8	<i>Criando Funções</i>	61
9	<i>Entrada e Saída</i>	69
10	<i>Programas em Python</i>	73
11	<i>Tomada de decisão - i f</i>	77

12	<i>Funções Recursivas</i>	81
13	<i>Repetindo instruções com while</i>	85
14	<i>Tuplas</i>	91
15	<i>Listas - list</i>	95
16	<i>Dicionários</i>	101
17	<i>Conjuntos</i>	105
18	<i>Mutabilidade</i>	109
19	<i>Coleções e Sequências</i>	113
20	<i>Repetindo instruções com for</i>	115
	<i>II Conceitos Avançados</i>	119
21	<i>Tuplas Nomeadas - namedtuple</i>	123
	<i>Bibliografia</i>	125
	<i>Índice</i>	127

Lista de Figuras

1.1	Um outro tipo de python: uma Python Burmesa, foto de Susan Jewell, U.S. Fish and Wildlife Service (USFWS) ⓘ	17
1.2	Arquitetura simplificada um computador	18
1.3	Arquitetura simplificada de uma CPU	19
1.4	O interior de um chip, foto por Fritzchens Fritz ⓘ	19
1.5	Camadas de uma máquina virtual	20
1.6	Logo oficial de Python, a linguagem de programação	21
2.1	Exemplo de invocação do Python 3.4 a partir do prompt de comando do Windows 10	25
2.2	Exemplo de invocação do IDLE	26
2.3	Exemplo do Spyder no Windows 10	27
2.4	Exemplo Eclipse para Python no Windows 10	28
4.1	As posições de indexação das strings estão na verdade entre as letras	41
10.1	Programa do Tipo Define-Processa-Escreve	73
10.2	Programa do Tipo Lê-Processa-Escreve	74
10.3	Programa do Tipo Lê-Processa-Escreve-Repete	74
15.1	Posição dos índices em listas	96
18.1	Primeiro passo, definir a como uma lista	109
18.2	Na memória podemos ver que o nome aponta para uma lista	110
18.3	Segundo passo, definir b como uma lista de zeros	110
18.4	Uma nova lista é criada para b	110
18.5	Terceiro passo, definir colocar a dentro de b	110
18.6	A lista a agora está dentro da lista b. Veja que ela não é copiada para dentro, mas apontada por uma referência. O programador não pode acessar essa referência e vê a lista b como contendo uma lista	111
18.7	Quarto Passo, alterar a	111
18.8	Agora quando alteramos a, o efeito é sentido em b.	111
18.9	Quinto passo, imprimir b, vemos que a mudança em a se reflete em b.	111

Lista de Tabelas

4.1	Tipos numéricos de Python	32
4.2	Operadores para números e sua prioridade, quanto mais baixo na tabela, maior a precedência, i.e., as operação são realizadas antes.	33
4.3	Escrevendo números em outras bases, com exemplo para o número 18 em base 10	33
4.4	Operadores lógicos bit a bit	34
4.5	Funções matemáticas incorporadas a linguagem	37
4.6	Operadores relacionais	38
4.7	O operador lógico not	44
4.8	Tabelas verdade para operadores lógicos and e or	44
4.9	Operadores de Python e sua prioridade.	46
4.10	Operações diretas nas próprias variáveis	46
7.1	Alguns Módulos de Python	55
9.1	Opções do comando print	69
17.1	Operadores de conjuntos para elemento e e conjuntos A e B	106
19.1	Funções e operadores aplicáveis a sequências s e t, item x e inteiros i, j e cfk	113

*Dedicado aos alunos e alunas que fazem
as perguntas certas e, mais ainda, aos que
fazem as perguntas erradas.*

PRIMEIRO, UM AVISO, ESTE LIVRO É UMA OBRA EM CONSTRUÇÃO. O livro, suas partes, seus capítulos e seções podem estar incompletos e inacabados. Certamente ainda estão fora de ordem e alguns capítulos misturam conceitos simples com conceitos necessários que só seriam necessários mais tarde.

Se você teve acesso a uma cópia desse livro, é porque foi escolhido pelo autor. Ainda não é hora de reclamar, mas é hora de dar sugestões.

Parte I

Conceitos Iniciais

1

Introdução

ESSE LIVRO É SOBRE PYTHON. Não sobre as cobras, é claro. Python é uma linguagem de programação muito usada atualmente¹ e extremamente prática para o ensino de programação e também para o uso em programas reais.

Esse livro é fruto da experiência de alguns anos ensinando Python na Universidade Federal do Rio de Janeiro, para alunos do primeiro período de diferentes Engenharias, e alguns repetentes, além do uso de Python para realizar trabalhos de pesquisa. Muito do que aprendi inicialmente veio do curso preparado pelos professores Paulo Roma e Claudio Esperança, que está disponível na rede <http://orion.lcg.ufrj.br/python/>. Também tenho que agradecer as aulas preparadas pelo grupo LadyBug <http://www.ladybugcodingschool.com/index.html>, que ajudaram a formar uma ideia de como a programar em Python pode ser aprendido.

1.1 Um pouco sobre computadores

COMEÇAMOS A ENTENDER O COMPUTADOR pelas suas características físicas. Porém, vamos estar mais interessados em suas características abstratas e no que chamamos de computador virtual, ou seja, o computador abstrato que é definido pela forma como programamos.

1.1.1 Principais partes de um computador físico

O computador pode ser dividido em 4 partes principais: unidade de processamento central (ou CPU), memória principal, memória auxiliar e dispositivos de entrada e saída.

1.1.2 A CPU

A UNIDADE DE PROCESSAMENTO CENTRAL, OU CPU (do inglês Central Processing Unit) é a parte do computador responsável por

¹ Em fevereiro de 2016 Python ocupava o quinto lugar entre as linguagens mais usadas no Tiobe Index http://www.tiobe.com/tiobe_index?page=index



Figura 1.1: Um outro tipo de python: uma Python Burmesa, foto de Susan Jewell, U.S. Fish and Wildlife Service (USFWS) ④

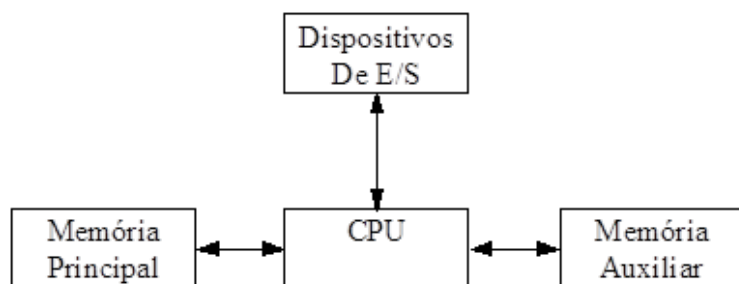


Figura 1.2: Arquitetura simplificada um computador

controlar todo o equipamento. Ela pode ser dividida entre Unidade de Controle, que faz o controle propriamente dito, e Unidade Lógica e Aritmética (ALU), responsável pelas operações lógicas e aritméticas que devem ser realizadas pelas instruções.

É a CPU que realiza o ciclo básico do computador: ler a instrução a ser executada da memória, interpretar a instrução, ler os dados necessários da memória, executar a instrução e escrever os dados na memória.

Também é a CPU que controla também a via de transmissão dos dados entre os dispositivos de entrada e saída, as memórias principais e secundária e a própria CPU. A esta via damos o nome de barramento de dados. O mecanismo de controle também “tráfega” em um barramento, ao qual damos o nome de barramento de controle.

O nome em inglês é data bus, o que já fez com que algum “tradutor especializado” cometam o pecado de chamá-la de ônibus! Dentro da CPU existe, em geral, uma pequena quantidade de memória, responsável por guardar o estado do computador. Chamamos esta memória de registradores. Alguns registradores são praticamente obrigatórios: o que guarda o endereço da instrução atual, o que guarda o endereço do topo da pilha de memória e um registrador de uso geral, utilizado para realizar as operações, conhecido normalmente como acumulador (ou registrador A, ou registrador 1), responsável por guardar o resultado das principais instruções. É comum que existam mais de um registradores de uso geral, como o acumulador.

É o tamanho e forma de uso do registrador de endereço que define o número de bytes de memória que um computador pode ter. Por exemplo, se o registrador de endereço tiver 16 bits, então o computador poderá endereçar $2^{16} = 65536$ posições de memória. Um Intel Core i7-4790 pode endereçar, por exemplo, 32GB de memória. Já a quantidade de bits característica de uma CPU é dada pelo tamanho dos registradores de uso geral. Logo se uma CPU tem 64 bits, isso indica que seus registradores de uso geral tem 64 bits. Normalmente o registrador de endereço é maior que os registradores de uso geral.

A arquitetura simplificada de uma CPU pode ser vista segundo a

Figura 1.3:

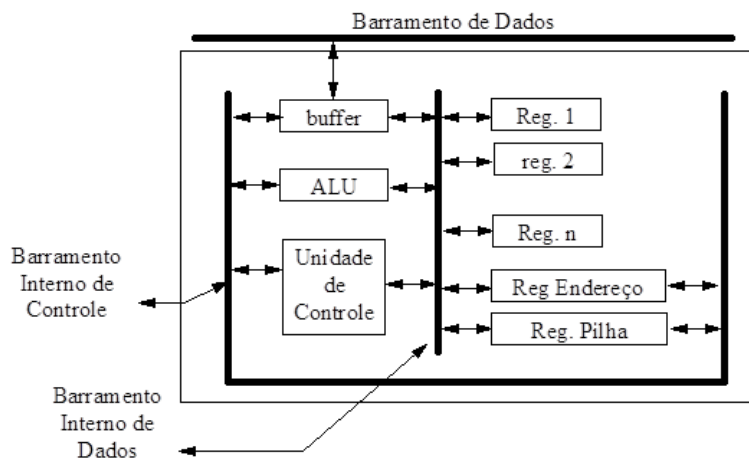


Figura 1.3: Arquitetura simplificada de uma CPU

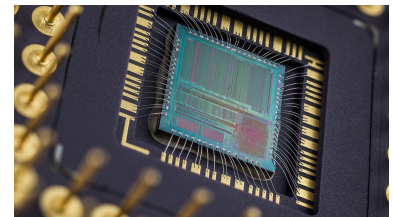


Figura 1.4: O interior de um chip, foto por Fritzchens Fritz ⓘ

1.1.3 A Memória Principal e Secundária

A memória principal de um computador é o local onde ficam guardados os dados e o programa, codificados na forma de 1's e 0's. As primeiras memórias utilizadas eram tubos de imagem ou dispositivos magnéticos. Atualmente a memória é feita por um tipo especial de circuito lógico conhecido como FLIP-FLOP, capaz de guardar a informação.

Inicialmente a memória dos computadores era muito pequena, na ordem de algumas dezenas de bytes. Em 2016 um computador pessoal pode ter Gigabytes de memória, sendo 4Gbytes ou 8Gbytes números bastante comuns.

A memória principal exige que o computador esteja ligado para funcionar. Para permitir guardar informação mesmo com o computador desligado, utilizamos a memória secundária, formada por dispositivo magnéticos ou ópticos.

A divisão de memória em principal e secundária permite que tenhamos uma memória extremamente rápida como memória principal e uma memória mais lenta como memória secundária. Alguns computadores possuem ainda uma memória mais rápida que a memória principal, e por isso bastante mais cara, conhecida como memória cache. A memória cache funciona fingindo ser uma parte da memória principal para a CPU, acelerando o acesso aos dados.

1.2 Os dispositivos de Entrada e Saída

Os dispositivos de entrada e saída tem como objetivo permitir que os dados e instruções sejam passados para o computador. Em um microcomputador principal dispositivo de entrada é o teclado. O

principal dispositivo de saída é o monitor ou vídeo.

Alguns dispositivos de entrada e saída são também dispositivos de memória secundária, como discos flexíveis e unidades de fita.

Outras unidades de entrada e saída são: placas de rede, microfones, leitoras de código de barra, leitoras de cartão, reconhecedores de caracteres, alto-falantes, fax, etc...

1.3 Entendendo Computadores Virtuais

COMPUTADORES SÃO MÁQUINAS PROGRAMÁVEIS DE USO GERAL QUE PROCESSAM INFORMAÇÃO. Relógios mecânicos também processam informação mas não são, programáveis, muito menos são de uso geral. Algumas máquinas são programáveis, como os teares que fazem tecidos, mas também não são de uso geral.

Quando usamos a palavra computador não queremos dizer exatamente esse objeto que é ligado a um teclado e a um monitor e que acabamos de descrever. É mais importante entender um computador como um conceito de algo que pode realizar computações. A maioria de nós não tem um contato direto com o computador construído de circuitos eletrônicos, CPUs ou memórias, mas sim com camadas e mais camadas de software que configuram um computador virtual.

O interpretador Python é uma máquina virtual que faz você usar o seu computador como se ele entendesse Python. O mesmo ocorre com outras linguagens. Ao usar Python você estará usando, virtualmente e na prática, um "Computador Python", que fica em cima de uma máquina virtual criada pelo seu sistema operacional, que fica em cima de uma máquina virtual criada pelo BIOS do seu computador, que fica em cima de uma máquina definida pela CPU do computador, e que também pode ser uma máquina virtual escrita em microcódigo sobre um hardware específico.

1.4 Algoritmos

UM COMPUTADOR É UMA MÁQUINA PROGRAMÁVEL CAPAZ DE EXECUTAR ALGORITMOS.

Um algoritmo é uma sequência de instruções passo a passo, possivelmente recursiva², capaz de resolver um problema. Para serem realizados, algoritmos tem que ser escritos em uma linguagem de programação.

Normalmente, nós esperamos que os algoritmos parem, isto é, acabem sua execução, em um número finito de passos, porém não é uma exigência obrigatória na prática. Podemos imaginar programas que funcionam para sempre, buscando resultados cada vez melhores, programas que nunca podem parar, como um programa de controle de voo ou mesmo programas que não fazem nada, como ciclos usados para gastar tempo da máquina. No

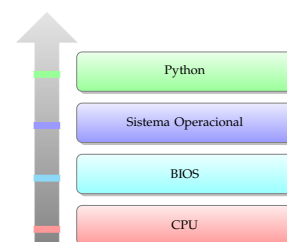


Figura 1.5: Camadas de uma máquina virtual

² Recursividade é a capacidade de uma função chamar a si mesma. A função fatorial, por exemplo, pode ser definida como $fat(1) = 1$ e $fat(x) = x \times fat(x - 1)$

nosso caso, trabalharemos apenas com algoritmos que param para permitir que sejam tiradas algumas conclusões³.

O objetivo dos programas de computador é realizar alguma transformação sobre um conjunto de dados. Se estivermos interessados em programas realmente úteis, devemos exigir alguma saída de dados do programa. Em geral, os programas exigem uma quantidade de informação, também na forma de dados, como entrada.

Para transformar os dados de entrada na saída desejada, devemos aplicar sobre eles os algoritmos. Para isso, muitas vezes usamos mais dados ainda, criados temporariamente⁴. Como queremos guardar, por algum espaço de tempo, esses dados, utilizamos a memória do computador.

1.5 A dificuldade de programar

PROGRAMAR É FÁCIL, SÓ QUE NÃO.⁵ A dificuldade de fazer um programa está muito ligada a dificuldade do problema em si. É fácil fazer um programa que calcula a média de dois números, mas é difícil fazer um programa que calcula uma estrutura de Engenharia Civil.

Programar, no nosso caso, é dar instruções para a máquina Python, assim não precisamos saber de muitas coisas que acontecem “por baixo dos panos”. Nosso foco será resolver problemas práticos de computação, isto é, como computar resultados que nos forneça uma nova informação a partir das informações definidas como dados de entrada⁶.

CADA PROGRAMA DE COMPUTADOR É COMO UMA RECEITA. Ele diz exatamente o que o computador deve fazer. As dificuldades de programar estão justamente em como dizer isso para o computador usando uma linguagem limitada, já que o computador não é capaz de preencher os vazios das suas ordens. É como se você estivesse ensinando alguém a fritar um ovo e tivesse que dizer que é necessário acender a chama do fogão, o que lhe parece óbvio. Outra dificuldade está em conseguir organizar suas ordens em passos lógicos utilizando essas linguagens limitadas.

1.6 Pequena História

PYTHON FOI CRIADA POR GUIDO VAN ROSSUM no final da década de 80⁷. O nome não é em homenagem as cobras, mas uma homenagem ao grupo britânico de comediantes *Monty Python*, porém o símbolo acabou adotando duas “cobrinhas” e a linguagem é hoje mais associada a elas do que ao grupo. A linguagem é totalmente aberta e gratuita. A quase totalidade dos pacotes adicionais que tornam Python tão interessante são também grátis.

³ Um erro típico de programação é fazer um programa que devia parar, mas não para.

⁴ Por exemplo, para calcular a área de um círculo temos que ter o valor de π

⁵ Programar é como criar encantamentos que fazem com que um gênio escondido no computador processe a informação.

⁶ Por motivos didáticos, apesar de tratar de problemas práticos, também limitaremos a dificuldade dos mesmos

⁷ Você pode saber mais sobre a história da linguagem em https://en.wikipedia.org/wiki/History_of_Python



Figura 1.6: Logo oficial de Python, a linguagem de programação

A referência básica da linguagem, em constante evolução, é o site oficial: <https://www.python.org/>.

1.7 O que é Python

PYTHON É UMA LINGUAGEM DE PROGRAMAÇÃO interpretada, de tipos fortes e dinâmicos, com uma sintaxe simples e clara e com muitas bibliotecas, ou módulos como se diz na cultura Python, que permitem tratar dados de forma eficiente.

Ser interpretada significa que os programas em Python não são convertidos em linguagem de máquina, mas sim executados diretamente por outro programa, o interpretador.⁸ Na versão default e mais usada de Python o interpretador é construído em C e é chamado de CPython.⁹

Ser de tipos fortes significa que os valores em Python são de um tipo único e imutável, como o tipo dos Inteiros, e por dinâmico significa que os nomes podem assumir os valores ao longo do programa¹⁰, e não precisam ter o mesmo tipo de valor o tempo todo.¹¹

A sintaxe de Python é minimalista. Ao mesmo tempo ela ajuda o programador a escrever programas corretos e visualmente elegantes, fornece uma quantidade pequena de formas de escrever seus comandos. A sintaxe permite que um ser humano, ou uma máquina, entenda mais facilmente o que está sendo programado. Mesmo assim, programas de computador podem parecer crípticos a um iniciante.

Um programa simples em Python, por exemplo, tem a seguinte forma:

```
def fatorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n*fatorial(n-1)
```

```
print(fatorial(5))
```

O resultado de sua execução será:

```
120
```

1.8 Que versão?

PYTHON É UMA LINGUAGEM COM DUAS VERSÕES, que diferem um pouco na sintaxe. As versões são numeradas em 2 e 3. A versão 2 é ainda utilizada porque muitas ferramentas bastante complexas contruídas em Python ainda são baseadas nela¹². A versão 3 é a versão atual.

⁸ Programa são executados diretamente em interpretadores ou convertidos em linguagem de máquina por compiladores. Compiladores normalmente produzem código mais eficiente, interpretadores normalmente são mais fáceis de usar.

⁹ C é outra linguagem de programação, de tipos fracos e estáticos, compilada, altamente eficiente mas com uma sintaxe que permite ao programador fazer muitos erros sutis, difíceis de encontrar.

¹⁰ Na tipagem dinâmica o tipo de uma variável não pode ser determinado por análise estática, isso é, ele só pode ser determinado em tempo de execução. Usualmente, linguagens com tipagem estática exigem que uma variável seja declarada antes de ser usada e que seu tipo seja definido de forma definitiva

¹¹ A principal diferença é que em Python você **não** precisa declarar uma variável antes de usar.

¹² As versões são tão parecidas que a maioria do código executado na geração desse livro está na verdade sendo lida por um interpretador Python 2.

A versão 2 deve parar sua evolução na subversão 2.7, que será suportada até 2020. Enquanto isso, a versão 3 deve continuar a evoluir¹³. No momento em que esse livro começou a ser escrito, a versão oficial era a 3.5 e a versão 3.6 já estava em fase de testes.

Se você quer usar a versão 2 de Python com esse livro, a principal diferença está no uso da função `print`, que exige parênteses, em vez do comando `print`, que não exige. Uma maneira de usar a função `print` é colocar em cada programa seu a linha:

```
from __future__ import print_function
```

1.9 *Uso de números nas listagens*

Algumas vezes nossos programas terão uma numeração das linhas de código. Isso não faz parte do programa, mas serve para ajudar a entender uma explicação.

```
1 def fatorial(n):
2     if n == 0 or n == 1:
3         return 1
4     else:
5         return n*fatorial(n-1)
6
7 print(fatorial(5))
```

¹³ O site oficial diz que Python 2 é legado, Python 3 é o presente e futuro da linguagem

Todos os programas nesse livro foram rodados tanto em Python 3 como em Python 2, usando essa linha de código

2

Usando o Interpretador Python

2.1 Como obter uma versão de Python

Para usar Python você provavelmente terá que instalá-lo em seu computador. A boa notícia é que existe uma versão gratuita para você.

2.1.1 No Windows ou no Mac

Siga os seguintes passos:

-
1. Acesse o site <https://www.python.org/>
 2. Clique em Download (na barra horizontal principal)
 3. Clique no botão da última versão de Python (Python 3.5.1)
 4. Salve o arquivo
 5. Execute o arquivo para instalação
-

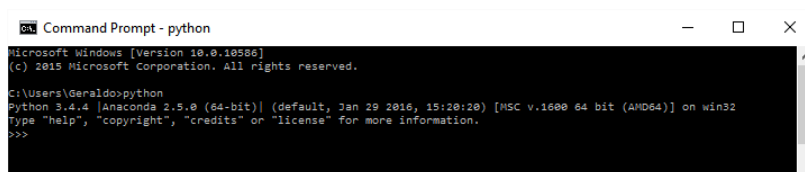
2.1.2 No Linux (Ubuntu)

A maneira mais fácil é usar o pacote apt-get e instalar o interpretador python3 e ainda o editor que vamos usar nesse livro como exemplo. O comando a ser dado é:

```
sudo apt-get install python3 idle3
```

2.2 Invocando Python

Ao ativar o comando python ou python3 em qualquer terminal, console ou prompt de comando, você receberá algo como:



```
Microsoft Windows [Version 10.0.10586]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\gerald\python
Python 3.4.4 [Anaconda 2.5.0 (64-bit)] (default, Jan 29 2016, 15:20:20) [MSC v.1600 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Figura 2.1: Exemplo de invocação do Python 3.4 a partir do prompt de comando do Windows 10

Você pode usar outras interfaces para o interpretador Python. Existem ambientes de desenvolvimento que tentam substituir o Matlab, como o Spyder, ambientes dentro do Eclipse e um ambiente simples e fácil de usar que se chama IDLE e vem com a distribuição default de Python para Windows e MacOS.

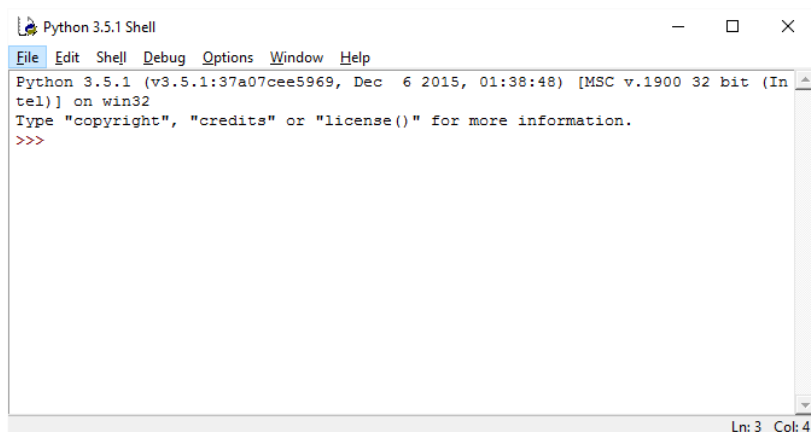


Figura 2.2: Exemplo de invocação do IDLE

Todo arquivo Python deve ter a extensão “.py”.

2.3 Ambientes na Rede

É possível programar direto na rede nos sites (entre outros):

- **Ideone** Um ambiente para múltiplas linguagens de programação que suporta Python 2 e 3. <https://ideone.com/>
- **Python Tutor** Um ambiente para Python que mostra o que está acontecendo na memória. Útil para entender a representação interna de Python. Permite sessões compartilhadas. <http://www.pythontutor.com/>
- **Skulpt** Python implementado no navegador. Pode ser baixado para você ter o código em sua máquina. <http://www.skulpt.org/>
- **Python Anywhere** Python on-line com o interpretador IPython <https://www.pythonanywhere.com/>
- **Repl It** Várias linguagens disponíveis on-line. <https://repl.it/>

2.4 Python para Smartphones

Também é possível encontrar versões de Python para smartphones. Abaixo duas versões gratuitas, existem outra pagas.

- **QPython** Python para Android. <http://qpython.com/>
- **Python For iOS** Python para iOS. <http://pythonforios.com/>

2.5 Ambientes mais poderosos

Existem outros ambiente Python mais poderosos. No início eles são tão poderosos que podem atrapalhar, mas eu recomendo que mude para eles com o tempo.

2.5.1 Anaconda

Anaconda é uma super-distribuição de Python que na prática compete com o Matlab. Ela usa vários interfaces como Spyder e IPython.

2.6 Spyder

Esse é um ambiente super poderoso para uso do Python, que lembra também o Matlab. Uso recomendado.

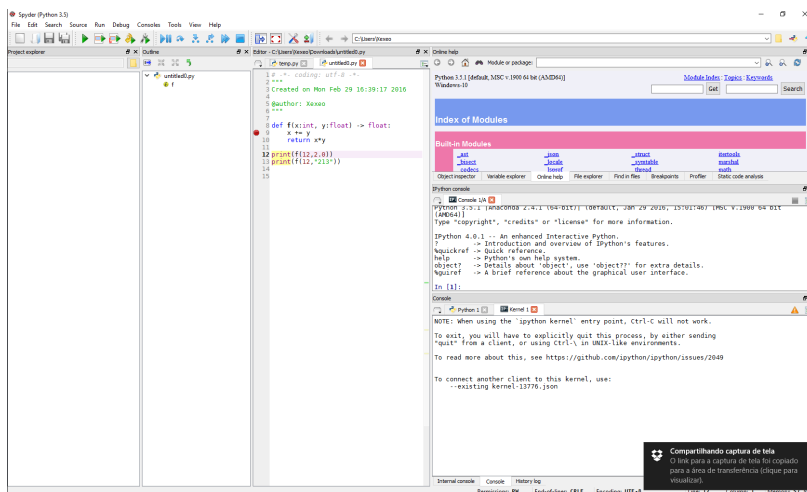


Figura 2.3: Exemplo do Spyder no Windows 10

2.7 Eclipse for Python

O Eclipse é bem pesado, mas é provavelmente o melhor ambiente de desenvolvimento disponível para várias linguagens. Se você usa mais de uma linguagem, principalmente no Linux, é sua escolha provável.

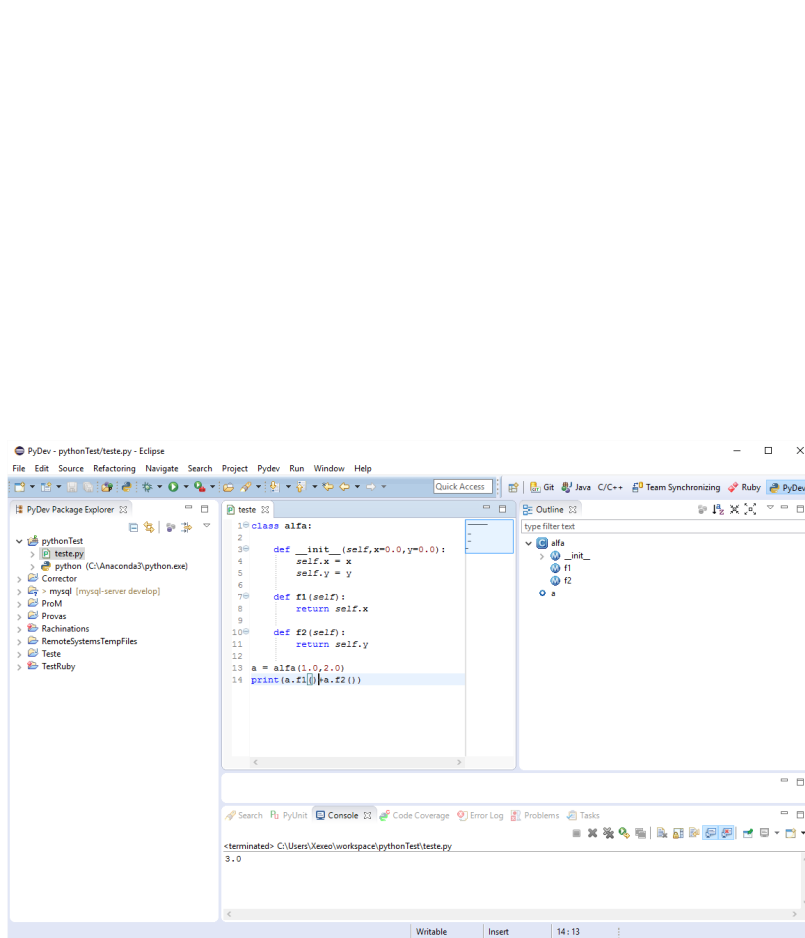


Figura 2.4: Exemplo Eclipse para Python no Windows 10

3

Python como Calculadora

É POSSÍVEL PODE USAR O INTERPRETADOR PYTHON COMO UMA CALCULADORA. Para isso basta escrever qualquer operação que deseja e mandar executar diretamente.

Essa é a vantagem de um interpretador. Ele executa imediatamente o que digitamos. Operações, como as operações aritméticas podem ser feitas imediatamente.

```
>> 2+4+6  
  
12  
  
>> 9*5+4/6.0  
  
45.6666666667
```

Também podemos digitar programas simples, como o famoso programa que imprime “Hello World!”¹.

```
>>print("Hello_World!")
```

Gera a resposta:²
Hello World!

Enquanto um programa levemente mais complicado:³

```
>>x = 1  
y = 3  
for i in range(5)  
    x += y  
    print 'somando_', y, '_a_', x  
print( 'Resultado_=_',x)
```

Gera a resposta somando 3 a 1

somando 3 a 4
somando 3 a 7
somando 3 a 10
somando 3 a 13
Resultado = 16

¹ print() é uma função que imprime o valor de seus argumentos no monitor. Mais tarde formalizaremos seu funcionamento

² Esses programas estão sendo executados diretamente no processador de texto sendo usado, o L^AT_EX, de modo que os resultados impressos são calculados na hora que o documento é gerado. A única diferença é que os programas executados são levemente modificados para a versão 2.7 do Python, e contém alguns comandos L^AT_EX para ficarem mais bonitos. Isso acontece porque essa é a versão integrada ao ShareL^AT_EX, mas não a que estamos aprendendo aqui.

³ Não importa se você não entender o programa agora, apenas digite e veja o que acontece!

3.1 Comentários

Comentários são partes de um programa que não são considerados pelo interpretador. Servem, justamente, para comentar nosso código.

Em Python um comentário começa com `#`. Tudo que vem depois, na mesma linha, é desconsiderado.

3.2 A instrução *pass*

A instrução `pass` não faz nada. Ela deve ser usada em lugares onde precisaríamos obrigatoriamente ter uma instrução mas não sabemos, ou não queremos colocar uma ali naquele instante.

Normalmente ela não aparece em programas terminados, mas é útil durante a construção de um programa poder fazer pedaços de código que podem ser executados e nada fazem.

```
# Esse programa não faz absolutamente nada  
pass
```


4

Tipos de Dados

COMPUTADORES PROCESSAM INFORMAÇÕES. No mundo real, os seres humanos também processam informações, normalmente adquirindo-as por meio de seus sentidos. Ouvimos sons, enxergamos imagens, sentimos pressão na pele, gostos e cheiros.¹

Um computador, real ou virtual, não processa as informações como seres humanos. Na prática, o que ele processa são dados, ou seja, representações das informações que temos. Números, por exemplo, estão guardados na memória do computador em uma representação de base 2, e não na base 10 como usamos normalmente. Já caracteres (como a letra 'A') são codificados internamente como números e só são transformados no símbolo que conhecemos como 'A' no processo de escrita. Finalmente, outras formas de informação, como sons, imagens, filmes, etc. são representados de forma mais complexa, de acordo com padrões combinados internacionalmente ou ainda formatos proprietários de empresas.

Enquanto na Matemática trabalhamos com conjuntos (sem tipos), na Computação é muitas vezes mais adequado, ou mais fácil, criar Tipos que permitem entender melhor o que é possível fazer, ou não, com um valor.

¹ As informações que são processadas em um computador geralmente estão codificadas de alguma forma. A forma mais simples de entendermos são os números inteiros. No computador, os números inteiros são representados como sequências de 1s e 0s.

4.1 O que é um Tipo de Dados

SIMPLIFICANDO, UM TIPO DE DADOS, OU SIMPLEMENTE UM TIPO, É UM CONJUNTO DE VALORES QUE TEM ALGUMA PROPRIEDADE. Por valores dizemos algo como 1, 301.05 ou "alfa". Tipos de Dados escondem do programador como os dados são representados dentro da linguagem, fornecendo uma interface externa adequada [Cardelli and Wegner, 1985].

Entre as propriedades dos Tipos estão seu comportamento, por exemplo, que operações podemos fazer com eles, sua representação interna e externa e também a relação entre os Tipos. Alguns Tipos, por exemplo, englobam outros Tipos (dizemos que são super-tipos dos sub-tipos).

Mais tarde veremos que em Python os Tipos são todos Classes. Há uma sutil diferença entre Classes, mas ela é irrelevante em

Python: para muitos autores um tipo é uma espécie de interface ou protocolo que pode ser implementado por várias Classes. Em Python, Classes e Tipos são a mesma coisa².

4.2 Os Tipos Numéricos

EM PYTHON, UMA DAS FORMAS MAIS BÁSICAS DE MANTER INFORMAÇÃO SÃO NÚMEROS. Python possui três Tipos numéricos, que buscam representar no computador os conjuntos dos Inteiros (\mathbb{Z}), dos Reais (\mathbb{R}) e dos Complexos (\mathbb{C}). A Tabela 4.1 mostra esses tipos.

Tipo	Significado	Exemplo
int	os Inteiros \mathbb{Z}	1, -2, 10012031203
float	aproximações dos Reais \mathbb{R}	1.0, 2e10, 1123.123123
complex	aproximações dos Complexos \mathbb{C}	10.0 + 4.5j

Por que diferenciar Inteiros, Reais e Complexos? Basicamente isso se dá por causa da forma como são usados e das necessidades de velocidade e armazenamento. Números inteiros podem e devem ser rapidamente processados e são usados muitas vezes para controlar a lógica dos programas, enquanto podemos trocar um pouco dessa velocidade pela capacidade de trabalhar com números Reais, que exigem mais complexidade na representação. O mesmo se dá com números Complexos.

4.3 Os Inteiros

EM PYTHON, O TIPO DE DADO QUE EQUIVALE AO QUE CHAMAMOS DE CONJUNTO DOS INTEIROS (\mathbb{Z}) na Matemática são os int³.

Os int fornecem operadores que permitem somá-los, subtraí-los, multiplicá-los, dividi-los ou ainda calcular o resto da divisão. Temos também funções que recebem números, inclusive inteiros, e fornecem algum resultado, inteiro ou não, como abs. Veremos que existem outras funções associadas aos int que nem sempre são típicas dos inteiros como os conhecemos na Matemática, mas são necessárias ou interessantes para que eles funcionem em Python.

Outra diferença básica da Matemática é que, em Python, um inteiro (int) não é um real (float), mas ambos são números⁴.

As principais operações com int, e outros tipos de números, são mostradas na Tabela 4.2, que inclui a precedência dos operadores.

Em expressões, como $4 + 5 * 3$, quanto mais alto o valor da precedência, antes a operação é feita. No caso, a resposta para a expressão é $4 + (5 * 3) = 19$ e não $(4 + 5) * 3 = 27$. No caso de operações de mesma precedência, elas são realizadas da esquerda para direita.

Algumas operações feitas com int:

```
>>> 10**2+2*4%3
```

² Em algumas linguagens de programação o nome Tipo é usado para Tipos Primitivos da linguagem. Já foi o caso de Python, mas isso mudou a partir da versão 2.2.

Tabela 4.1: Tipos numéricos de Python

³ Python 2 apresenta dois tipos de inteiros, curtos e longos. A implementação atual unificou os dois

⁴ Mais especificamente, int não é uma subclasse, ou subtipo de float, mas ambos são subclasse de Number

102

>> 1000*18//6

3000

Operador	Função
+, -	soma e subtração
*, /, //, %	multiplicação, divisão, divisão inteira e resto
+, -	positivo, negativo (unários)
**	exponenciação
()	parênteses de precedência

O operador % calcula o resto da divisão. Ele também é conhecido como operador módulo, como essa operação é conhecida na matemática. O operador // calcula a divisão inteira, isso é, quantas vezes o divisor cabe no dividendo.

4.3.1 Como escrever inteiros

Inteiros podem ser escritos dentro de um programa nas bases 10, como usamos normalmente, mas também nas bases 2, 8 e 16, como é comum usar em alguns problemas de computação. Para isso se usa um prefixo, como mostrado na Tabela 4.3. A tabela também apresenta que função usar para escrever um número dado na base 10 em uma dessas bases padrão da linguagem.

Base	Prefixo	18 ₁₀	Função para escrever na base
2	0b	0b10010	bin(x)
8	0o	0o22	oct(x)
16	0x	0x12	hex(x)

Python sempre escreverá o número em base 10, a não ser que você use uma das funções específicas da Tabela 4.3. Por exemplo.

>> hex(18)

0x12

>> 0b10101

21

>> bin(0b10101)

0b10101

4.3.2 Operadores bit a bit

Os números inteiros, tipo *int*, podem ser facilmente escritos em base 2. Na computação existem alguns operadores tradicionais que são usados com inteiros em base 2, que são a negação, o *ou*, o *e*, o *ou-exclusivo* e o deslocamento.

Tabela 4.2: Operadores para números e sua prioridade, quanto mais baixo na tabela, maior a precedência, i.e., as operação são realizadas antes.

Como veremos mais adiante, além de servir para fazer contas, o tipo *int* é bastante usado no controle de quantos passos um programa deve ter e em contagens.

Tabela 4.3: Escrevendo números em outras bases, com exemplo para o número 18 em base 10

Para escrever em outras bases além de 2, 8 e 16 você pode usar o módulo `numpy.base_repr`, porém para pegar uma string e converter para um número, considerando que a string está em uma base específica basta usar a função `int< string >, < base >`

Os operadores bit a bit, na prática, só fazem sentido quando vistos na representação em base 2. Se não conhece bases numéricas, procure um texto que ajude a compreender o conceito

Valor1	Valor2	~Valor1	Valor1 & Valor2	Valor1 Valor2	Valor1 ^ Valor2
0	1	1	0	0	0
0	0	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Tabela 4.4: Operadores lógicos bit a bit

Os operadores lógicos bit a bit (&,|,~) funcionam aplicando a comparação lógica bit a bit. As tabelas verdade para quando aplicados a um bit são:

Esses operadores normalmente são usados para trabalhar com mais de um bit.

Uma expressão usando esses operadores⁵:

```
>> bin(0b10101100 & 0b11110000)
```

Que resulta em:

```
0b10100000
```

Os operadores de deslocamento, como diz o nome, deslocam os bits para esquerda ou para direita, como no exemplo a seguir.

```
>> print(bin(0b01001101011 << 2))
```

```
0b100110101100
```

```
>> print(bin(0b01001101011 >> 3))
```

```
0b1001101
```

Note nesses exemplos que apesar de Python entender números na base 2 (e 8 e 16 também) ele só é capaz de imprimi-los com o uso dos comandos específicos mostrados na Tabela 4.3

4.3.3 Alguns Problemas com Inteiros

Exercício 1. João precisa fazer um muro com 81 tijolos de extensão e 19 tijolos de altura. A loja em que ele faz compras, porém, só vende tijolos a dúzia. Quantos pacotes de 12 tijolos João deve comprar e quantos tijolos sobrarão?

```
1 19 * 21 // 12
2 19 * 21 % 12
```

Calculamos na linha 1 a quantidade de pacotes, usando a divisão inteira para receber um número inteiro. Na linha dois calculamos a sobre usando o operador de resto.

Exercício 2. José saiu de casa para ir ao trabalho às 6 horas e 35 minutos. Normalmente José chegaria ao trabalho 1 hora e 2 minutos, porém devido ao trânsito chegou às 9 horas e 44 minutos. Quantos minutos a mais levou José?

```
1 (9*60+44) - (6*60+35+1*60+2)
```

⁵ Para obter um resultado em binário é obrigatório usar a função `bin`, caso contrário há a impressão automática na base 10

Nesse programa convertemos todas as medidas para minutos.

Exercício 3. Verifique se o número 625 é um quadrado perfeito

```
sqr(625)*sqr(625)
```

Como ainda temos poucas ferramentas, essa é uma forma simples de fazer essa verificação.

4.4 Os Reais, ou Quase Isso

HÁ UMA DIFERENÇA MUITO GRANDE ENTRE INTEIROS E REAIS PARA O MUNDO DA COMPUTAÇÃO. Enquanto qualquer inteiro pode ser representado por uma quantidade finita de algarismos, mesmo que ela seja muito grande, existem números reais cuja representação é infinita. Um exemplo disso é o número π , cuja representação inicial é 3.141692.... Dentre esses números, alguns são racionais⁶, e poderiam ser representados por dois inteiros, porém existem ainda os Irracionais, que não pode ser escritos na forma de fração.

Já que o problema de representar reais é, na prática, intratável, a solução aceita, que faz uma bom balanço de utilidade e velocidade, é escolher uma representação de tamanho fixo (binário) para os números reais. Essa notação é incapaz de representar todos os números, porém deve poder se aproximar suficientemente de qualquer número útil. A questão de representar números reais é bastante importante e exige a escolha de dois parâmetros principais: qual a maior quantidade possível de números significativos e qual o melhor expoente.

Como os números definidos em Python, e em todas linguagens de programação, não são realmente reais, são conhecidos como números de ponto flutuante, ou simplesmente float. Em Python são usados apenas números de dupla precisão⁷. Outras linguagens usam duas representações, uma simples e uma dupla, mas a sobrecarga de trabalho que isso traria ao processamento de números de ponto flutuante em Python acabou levando a escolher apenas o mais preciso.

Para saber qual a precisão disponível no seu interpretador Python use o nome `sys.float_info`, que permite perguntar vários limites dos floats⁸.

Enquanto esse livro é escrito, aqui estão os limites do interpretador Python em uso, onde Epsilon é a o valor entre 1.0 e o menor número maior que 1.0⁹:

Maior positivo: `sys.float_info.max = 1.79769313486e+308`

Menor positivo: `sys.float_info.min = 2.22507385851e-308`

Bits na mantissa: `sys.float_info.mat_dig = 53`

Epsilon: `sys.float_info.epsilon = 2.22044604925e-16`

Maior quantidade de algarismos significativos na base 10:

`sys.float_info.dig = 15`

⁶ O conjunto dos números racionais, \mathbb{Q} , é o conjunto dos números que podem ser representados na forma de uma fração entre dois inteiros. Alguns reais não são racionais, como $\sqrt{2}$, conhecidos como irracionais. Dentro dos irracionais ainda existem os transcendentais, que não podem ser expressados como uma raiz de uma equação de um polinômio com coeficientes racionais, como π .

⁷ Mais sobre essa questão em Python está explicado em <https://docs.python.org/3.5/tutorial/floatingpoint.html>

⁸ Mais informações em <https://docs.python.org/3.5/library/sys.html>

⁹ A função `print` será vista detalhadamente no Capítulo 9

Os float podem fazer qualquer operação numérica, como por exemplo:

```
>> 10.5**(2.4+1.0)*2.3
6819.79708383
```

A operação float de resto retorna também um float. Ela funciona com float dos dois lados. O programa¹⁰

```
a = 10.3
b = 2.5
c = a / b
d = a // b
e = a % b
f = e - 0.3
print(a,b,c,d,e,f)
```

Resulta em:

```
10.3 2.5 4.12 4.0 0.30000000000000007 7.21644966006e-16
```

Nesse resultado, os dois últimos números mostram um erro justamente por causa da questão de representação.

Outra informação importante: quando float são misturados com int toda expressão é convertida para float. Se o int for muito grande, ou seja, tiver mais de `sys.float_info.dig` algarismos, haverá perda de precisão.

```
a = 123456789123456789123456789
b = float(a)
print(a,b)
```

Resulta em:

```
123456789123456789123456789 1.23456789123e+26
```

¹⁰ Esse programa usa variáveis e a instrução de atribuição, caracterizada pelo símbolo de igual =. Isso será visto no capítulo 5.

4.4.1 Formas de escrever números de ponto flutuante

Para um número ser do tipo float ele deve ser escrito de uma forma específica, contendo o ponto decimal, como em 10.4. Os números antes e depois do ponto decimal podem ser deixados em branco, como em 1. ou .1. No caso de números negativos, basta colocar o sinal de negativo antes do número, assim a expressão `12.0+-10.0` é válida.

Uma outra maneira de escrever números de ponto flutuante é usar a notação científica. Para isso, o número é escrito com uma mantissa e um expoente, separados pela letra e. São válidos os números `-12.4e13`, que significa -12.4×10^{13} , ou `.4e3` que é o número 400.0.

4.4.2 Alguns Problemas com Reais

Exercício 4. Calcule a média de 3 números reais: 10.0, 5.0, 5.8.

$(10.0+5.0+5.8)/3.0$

Exercício 5. Um automóvel partiu de São Paulo para o Rio de Janeiro, fazendo a viagem em 6 horas. Sabendo que a distância rodoviária entre as duas cidades é de aproximadamente 442 km, calcula a velocidade média em m/s.

$442.0 * 1000.0 / (6.0 * 3600.0)$

Exercício 6. Sabendo que a Força Gravitacional entre dois objetos pode ser calculada pela fórmula $F_g = G \frac{M_1 \times M_2}{r^2}$, onde G é a constante de gravitação universal, $G = 6,67 \times 10^{-11} \frac{m^3}{kg s^2}$. Calcule a força de atração média entre a Terra e a Lua. A massa da Terra é aproximadamente $5,973332 \times 10^{24} kg$, a massa da Lua aproximadamente $7,3474271 \times 10^{22} kg$ e o distância média entre eles é 384.000 km.

$6.67e-11*5.973332e24*7.3474271e22/384000**2$

Nesse exercício usamos o fato que a operação de exponenciação acontece sempre antes das de multiplicação e divisão, e que essas ocorrem na ordem em que aparecem. Também usamos a notação científica. O resultado é: $1.9852505481772758e+26$, porém só devíamos nos preocupar com os três números mais significativos (por causa da limitação da distância Terra-Lua utilizada, lendo isso como 1.99×10^{26} ¹¹).

¹¹ Veremos mais tarde como controlar a precisão com que números são impressos

4.5 Usando Funções em Números

VÁRIAS FUNÇÕES ESTÃO DISPONÍVEIS EM PYTHON para ser usadas com números. Algumas delas fazem parte da linguagem propriamente dita, como `abs`, outras precisam ser primeiro incorporadas ao programa pela importação de módulos, como veremos mais adiante.

As funções matemáticas incorporadas, (*built-in*), são mostradas na Tabela 4.5:

Função	Significado
<code>abs(x)</code>	Valor absoluto do número
<code>complex(x)</code>	converte para complex
<code>float(x)</code>	converte para float
<code>int(x)</code>	converte para int
<code>max(x,y,z,...)</code>	calcula o máximo dos números
<code>min(x,y,z,...)</code>	calcula o mínimo dos números
<code>divmod(x,y)</code>	retorna 2 números, o quociente e o resto da divisão inteira

Tabela 4.5: Funções matemáticas incorporadas a linguagem

4.5.1 Problemas usando funções numéricas

Exercício 7. Ana, Beto e Carlos trabalham em uma loja. Eles tinham se comprometido a vender, respectivamente, R\$ 1.000,00, R\$1.250,00 e R\$2.450,00. Eles venderam, também respectivamente, R\$ 1.234,00 ,

A lista de todas as funções incorporadas a linguagem pode ser encontrada em <https://docs.python.org/3/library/functions.html>, outras funções são disponíveis com o uso de módulos, como veremos mais adiante

R\$ R\$ 1.534,38 e R\$2.970,45. Qual foi a maior diferença entre o valor vendido e o valor com que o vendedor se comprometeu?

```
max(1234.0-1000.0, 1534.38-1250.0, 2970.45-2450.0)
```

4.6 Operadores Relacionais

Outra necessidade comum é fazer comparações entre valores, se são maiores ou menores por exemplo, cuja resposta será verdadeiro ou falso. Isso é muito usado na computação, como veremos mais adiante.

Os operadores são os mesmos da matemática, mas devem ser escritos de uma forma específica. Todos tem a mesma precedência.

Operador	Uso	Significado
>	a > b	a é estritamente maior que b
<	a < b	a é estritamente menor que b
>=	a >= b	a é maior que ou igual a b
<=	a <= b	a é menor que ou igual a b
==	a == b	a é igual a b
!=	a != b	a é diferente de b
in	a in b	a está em b
not in	a not in b	a não está em b
is	a is b	a é a mesma coisa me memória que b
not is	a not is b	a não é a mesma coisa me memória que b

Os operadores >, <, >=, <= são bastante óbvios, mas devem ser escritos dessa forma, i.e., você **não** pode escrever `##/`

Chamamos a atenção para o operador de igualdade. Ele é escrito `==` e é totalmente diferente do operador de atribuição `=`¹².

Os operadores `in` e `not in` verificam se um valor está dentro de uma coleção de valores. Elas são importantes em Python, mas só serão vistas mais tarde.

Finalmente, os operadores `is` e `not is` verificam se um valor é exatamente o mesmo valor, na memória, que outro. Isso é uma espécie de igual mais limitado, pois não basta que os valores sejam iguais, mas exigem que seja algo como “a mesma cópia”. Isso sempre será verdade para valores iguais alguns tipos, como `int` e `bool`,

Vamos ver alguns exemplos, sendo que você deve prestar atenção ao último exemplo:

```
>> 10>5
True
>> 10<5
False
>> 10.0==5.0
False
>> 10.5 % 3.0 == 2.5
```

A resposta para essa conta é, em Python 3, 520.44999999999998. Isso também é um erro de representação por aproximação dos `float` e, tendo em vista as contas que foram feitas, deve ser interpretado como o número real 520.45.

Note que nesse problema, como é escrito em português, usamos a vírgula como separador decimal. Em Python, porém, lembramos que o separador decimal é o ponto. Também usamos o ponto em português para separar os milhares. Em Python não separamos os milhares.

Tabela 4.6: Operadores relacionais

¹² É extremamente comum o erro de programação de usar `=` quando se quer usar `==`, e o pior é que isso, algumas vezes, fornece uma instrução correta. Então, preste atenção a esse uso

Python tem duas grandes classes de tipos, os mutáveis e os imutáveis. Vamos estudá-los mais profundamente mais tarde. Ambos tem relação com o modelo de objetos. Um tipo mutável pode mudar de valor e continuar sendo o mesmo objeto, um tipo imutável não pode mudar de valor. Veja que a variável que guarda o valor de um tipo imutável pode mudar de valor, nesse caso indicando outro valor. Já uma variável que guarda o valor de um tipo mutável pode continuar indicando um mesmo objeto, mas esse objeto troca de valor. Os `int`, `float`, `bool` são imutáveis

False

Por que o último exemplo não fornece a resposta True? Nós já discutimos isso quando falamos dos float: a representação de ponto flutuante não é precisa. Assim, quando comparar números de ponto flutuante, e principalmente quando forem resultados de operação, nunca compare um número com outro, mas sim compare a diferença com um valor de precisão adequado. Por exemplo:

```
>> 10.5 % 3.0 - 2.5 < 1e10
```

True

4.7 Os Complexos, ou Quase Isso, De Novo

Complexos são escritos na forma $10.0 + 12.2j$, isto é, usando o *j* para indicar a parte imaginária. Internamente, números complexos são representados com dois float.

É possível pegar a parte imaginária e a parte real de um número complexo com as funções *real* e *imag*. Também é possível usar as funções *abs* e *conjugate*.

Para usar funções que usam complexos precisamos do módulo *cmath*, logo deixamos suas operações para mais tarde.

4.8 O Tipo String

ALÉM DE NÚMEROS, É MUITO COMUM QUE USEMOS O COMPUTADOR PARA PROCESSAR TEXTO. A maneira mais comum de representar texto é por meio do Tipo *str*, conhecido como *string*¹³.

Exemplos de string são:

```
"A"
'A'
"Uma frase"
'Uma frase'
""" Uma frase longa
    muito longa
    com várias linhas"""
```

Como vimos no exemplo acima, strings devem ser delimitadas por aspas (*"*), apóstrofes (*'*), ou três aspas quando queremos uma string que seja muito longa e use muitas linhas¹⁴.

Existem muitas funções que trabalham com strings no pacote *string*, e veremos elas mais tarde.

As strings podem ser concatenadas para criar outras strings. Para isso também se usa o operador *+*. Uma string vazia, sem caracteres, é indicada abrindo e fechando a string sem nada no meio, como em *""*. Uma string pode conter espaços, como em *" "*, uma string com um só espaço.

¹³ EM português muitas vezes usamos o termo sequência de caracteres, mas não há uma tradução direta e a palavra “string” será usada no texto

¹⁴ Muitas linguagens de programação possuem dois tipos, um de caracteres, ou seja, uma letra apenas, e um de strings, possivelmente sendo o segundo um vetor do primeiro. Em Python, não existe um tipo caractere. Uma string de uma letra é também uma string

O tamanho de uma string é dado pela função `len(s)`¹⁵. Essa é uma função muito usada, inclusive com outros tipos de dados.

Seguem alguns exemplos:

```
>> "alfa" + "beto"
alfabeto
>> "alfa" + "_" + "beto"
alfa beto
>> "alfa" + " " + "beto"
alfabeto
>> len("alfabeto")
8
>> len("a_l_f_a_b_e_t_o")
15
```

¹⁵ Nesse livro usamos uma marca especial para permitir que o leitor veja o espaço, ele não existe e não deve ser digitado no programa.

Não existe um operador para subtrair strings, mas existem funções que permitem modificá-las de várias formas, ou buscar strings dentro de strings.

Na seção 7.9 veremos mais funções sobre strings.

4.8.1 Índices de Strings

Um dos operadores mais poderosos sobre strings é o `[]`. Ele permite indexar um caracter específico da String ou pegar uma fatia da String. Ele também permite contar do início para o fim, ou do fim para o início (nesse caso usando números negativos).

A primeira letra de uma string está na posição zero.

```
>> "beto"[0]
b
```

```
>> "beto"[3]
o
```

Na contagem ao contrário, a última letra está na posição `-1`.

```
>> "beto"[-1]
o
```

```
>> "beto"[-3]
e
```

Como vemos, há uma pequena diferença na forma de contar. Ela existe por alguns motivos. Primeiro, não existe o número -0 , segundo, ela permite que o trabalho com fatias funcione melhor.

4.8.2 Fatias de Strings

FATIAS SÃO PEDAÇOS DE STRINGS.

Mais interessante ainda que poder pegar caracteres um a um, em Python podem pegar pedaços ou fatias de strings usando também o operador `[]`.

Primeiro vamos ver alguns exemplos mais simples, usando a notação início e fim da fatia, para frente e para trás. Para isso, usamos o operador fatia no formato `[<início>:<fim>]`.

```
>> "beto"[0:1]
```

```
b
```

```
>> "beto"[0:2]
```

```
be
```

```
>> "beto"[-3:-1]
```

```
et
```

Para entender como funciona devemos entender que as posições nas fatias, positivas ou negativas, não indicam verdadeiramente as posições, mas sim os intervalos entre as letras. A Figura 4.1 demonstra isso. Na figura devemos notar que a posição positiva final não pode ser usada em índices, apenas em fatias.

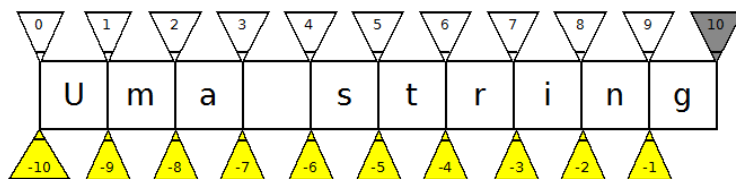


Figura 4.1: As posições de indexação das strings estão na verdade entre as letras

A notação de fatias também pode ser usada com 3 argumentos, como em `[<início>:<fim>:<passo>]`. Dessa maneira, podemos criar fatias especiais que pulam letras, como em:

```
>> "alfabeto_brasileiro"[1:12:2]
```

```
laeoba
```

Outra característica: se a posição inicial, ou final, é deixada em branco, então significa que queremos a fatia a partir do início, ou fim, da string.

```
>> "alfabeto"[:5]
alfab
```

```
>> "alfabeto"[5:]
eto
```

E se deixamos início e fim vazios uma cópia da string é criada.

```
>> "alfabeto"[:]
alfabeto
```

Finalmente podemos colocar tudo junto:

```
>> "alfabeto_brasileiro"[::2]
afbtrsliao
```

Também podemos usar passos negativos, como em:

```
>> "alfabeto_brasileiro"[11:5:-1]
arb ot
```

Ou ainda calcular a inversão de toda a string com:

```
>> "alfabeto_brasileiro"[::-1]
orielisarb otebafla
```

4.8.3 O operador *in*

Strings, como todas as sequências, possui um operador *in* que diz se um elemento está dentro dela. Também é possível verificar se um elemento não pertence a uma string usando o operador *not in*.

```
>>> "1" in "1234"
True
>>> "5" in "1234"
False
>>> "1" not in "1234"
False
>>> "5" not in "1234"
True
```

4.8.4 Comparando Strings

A princípio, strings são comparadas em sua ordem lexicográfica, isto é, primeiro na "ordem alfabética", depois pelo tamanho da string.

Na verdade, cada caracter tem um valor inteiro, ou seu *Unicode code point*. Esse valor pode ser calculado com a função `ord(c)`, onde (c) é uma string de 1 caracter apenas. Também podemos calcular o caracter dado um número, com a função `chr(i)`, onde i é um número inteiro até 1.114.111, ou 0x10FFFF, o maior caracter Unicode.

Por necessidades de representação, "A" e "a" são caracteres diferentes para o computador.

O programa:

```
print(ord("a"))
print(ord("A"))
print("a"<"A")
```

Resulta em:

```
97
65
False
```

4.8.5 Convertendo String Numéricas em Números

Algumas strings contém valores numéricos, mas não podem ser usadas diretamente em contas. Para isso são usadas as funções de conversão de tipo que vimos na Tabela 4.5.

```
>> float("110.5")
110.5
```

4.8.6 Problemas com Strings

Exercício 8. Ana recebeu de Beto uma mensagem em código, onde cada terceira letra apenas deve ser lida. Se a mensagem recebida foi "asefduds grtrwewe jyacgmsdo", qual a mensagem original?

```
"asefduds_grtrwewe_jyacgmsdo"[2::3]
```

Veja que como as letras de uma string são contadas a partir do número zero, a primeira terceira letra é a de índice 2.

Exercício 9. Ana resolveu responder a Beto, usando novamente um código, porém agora eles combinaram inverter a string e depois escolher apenas as letras pares. Lembre, porém, que eles sempre começam contando a primeira letra como 1. A mensagem de Ana para Beto foi 'fohduaeryoemwatny sokhrnjertm wudEn'

```
'fohduaeryoemwatny_sokhrnjertm_wudEn'[::-1][1::2]
```

Nessa solução usamos dois passos. Como o operador `[]` acontece primeiro a esquerda, então primeiro invertemos a string para depois pegar as letras pares.

4.9 Booleanos

Uma das características típicas de programas de computador é tomar decisões. Algo será feito, ou não, caso uma condição seja verdadeira ou falsa. Para implementar os conceitos de verdadeiro ou falso foi criado o tipo `bool`, que possui apenas dois valores: `True` e `False`.

Com esses valores passamos a poder usar dois tipos de operadores: os operadores lógicos e os operadores relacionais.

4.9.1 Operadores lógicos

Os operadores lógicos implementam os operadores da Lógica Booleana, `not`, `and` e `or`. Apesar de nós usarmos as palavras *não*, *e* e *ou* na nossa linguagem de forma informal, a lógica booleana da significado fixos e formais para esses três operadores.

O operador `not` é unário, isso é, só tem um operando e nega o valor desse operando. Assim, `not True` tem o valor de `False` e `not False` tem o valor de `True`. Normalmente isso é indicado por uma tabela lógica, como a Tabela .

Valor	not Valor
True	False
False	True

Tabela 4.7: O operador lógico `not`

Os operadores `and`, que significa “e”, e `or`, que significa “ou” precisam de dois operandos, como em `True and False` ou `True or False`. Eles se comportam exatamente como os operadores lógicos, como mostra a Tabela 4.8.

Valor1	Valor2	Valor1 and Valor2	Valor1 or Valor2
False	False	False	False
False	True	False	True
True	False	False	True
True	True	True	True

Tabela 4.8: Tabelas verdade para operadores lógicos `and` e `or`

Em Python, expressões lógicas podem ser contruídas de qualquer tamanho, porém o `not` tem precedência sobre o `and`, que por sua vez tem precedência sobre o `or`.

Assim, `False and True or True` tem valor de `True`, pois o `and` acontece antes, resulta em `False`, que na expressão `False or True` resulta em `True`.

Vejamos alguns exemplos:

```
>> False and True or True
```

```
True
```

```
>> False and (True or True)
```

False

4.9.2 Valores verdade de outros tipos

Todo objeto em Python pode ter seu valor verdade testado.

Em Python, tem o valor verdade de falsidade

- None
- False
- o zero (0, 0.0, 0 + 0j) de qualquer tipo numérico
- strings ou sequências vazias, como "", [], (), {}
- instâncias de classes definidas pelo usuário, se a classe define métodos específicos, `__nonzero__()` ou `__len__()`, que retornam 0 ou False

Qualquer outra coisa é True.

Porém é importante perceber que apesar de 0, ou [], terem o valor verdade de falsidade, eles **não** são iguais a False na comparação `False==[]`.

4.10 Convertendo entre tipos

Python fornece funções para converter para os tipos numéricos: `int`, `float`, `bool`, que possuem os mesmos nomes dos tipos.

```
>> int(10.0)
10
```

Outras conversões entre tipos existem, mas devem ser vistas caso a caso.

4.11 Precedência de Todas os Operadores

A Tabela 4.9 apresentamos a lista de precedência completa de Python, como mostrado em <https://docs.python.org/3/reference/expressions.html>.

Expressões if-else analisadas no capítulo sobre if (11.1), expressões lambda analisadas mais tarde

4.12 None, o Tipo que não é nada

None é um valor e um Tipo que significa nenhum valor. Funções que não retornam nada são forçadas por Python a retornar None. Variáveis podem ser receber o valor None.

4.13 Operadores Aplicados a Própria Variável

Python possui uma família de operadores que aplica a si mesmo a operações. Por exemplo, `x += 1` soma 1 a x. Lembramos que a esquerda sempre deve aparecer uma variável, nunca um literal.

Operador	Função
<code>lambda</code>	expressão lambda
<code>if - else</code>	expressão condicional
<code>or</code>	ou lógico
<code>and</code>	e lógico
<code>not</code>	não lógico
<code>in, not in, is, is not, <, <=, >, >=, !=, ==</code>	operadores relacionais e de pertinência
<code> </code>	ou bit a bit
<code>^</code>	ou-exclusivo bit a bit
<code>&</code>	e bit a bit
<code>«, »</code>	operação de deslocamento
<code>+, -</code>	adição e subtração
<code>*, /, //, %</code>	multiplicação, divisão, divisão inteira, resto
<code>+x, -x, ~x</code>	positivo, negativo, negação bit a bit
<code>**</code>	exponenciação
<code>await x</code>	espera por x
<code>x[index], x[index:index], x(arguments...), x.attribute</code>	índice, fatia, chamada de funções, uso de atributos
<code>(expr...), [expr...], {key: value...}, {expr...}</code>	Binding ou uso direto de tupla, list, dicionários e conjuntos

Tabela 4.9: Operadores de Python e sua prioridade.

Operação	Significado	Operação	Significado
<code>a += b</code>	<code>a = a + b</code>	<code>a -= b</code>	<code>a = a - b</code>
<code>a *= b</code>	<code>a = a * b</code>	<code>a /= b</code>	<code>a = a / b</code>
<code>a //= b</code>	<code>a = a // b</code>	<code>a %= b</code>	<code>a = a % b</code>
<code>a **= b</code>	<code>a = a ** b</code>	<code>a &= b</code>	<code>a = a & b</code>
<code>a = b</code>	<code>a = a b</code>	<code>a ^= b</code>	<code>a = a ^ b</code>
<code>a <= b</code>	<code>a = a < b</code>	<code>a >= b</code>	<code>a = a > b</code>

Tabela 4.10: Operações diretas nas próprias variáveis

4.14 O que Sabemos Sobre Tipos de Dados

Python trabalha com valores que pertencem a tipos de dados. Os valores numéricos básicos são os `int`, `float` e `complex`. Ainda trabalhamos muito com `str`, para sequências de caracteres, `boolean` para os valores lógicos `True` e `False`, e ainda o tipo `None`, que indica a ausência de valor.

5

Variáveis e Atribuição

TODAS OS DADOS E VALORES USADOS EM UM PROGRAMA DE COMPUTADOR DEVEM ESTAR EM ALGUM LUGAR DA MEMÓRIA DO COMPUTADOR. Para se referir a esses valores usamos nomes que são chamados de variáveis.

Assim, quando escrevemos em nossos programas

```
x = 10.0
```

Estamos atribuindo a `x` o valor 10.0. Queremos dizer que o nome `x` passa a se referir ao valor 10.0¹. Normalmente dizemos que `x` vale, ou guarda, o valor 10.0.

Na linguagem da computação, `x` é uma variável. Esse nome vem do fato que o valor guardado em `x` pode variar ao longo do programa, mas apenas quando uma nova atribuição é feita². Por outro lado, `10.0` é um literal e uma constante. O valor de `10.0` é o mesmo, sempre.

Em Python, é mais correto ainda dizer que `x` é um nome que aponta para o valor 10.0. Todos os nomes apontam para algum valor. Para economizar memória, o interpretador CPython faz com que todos os nomes que apontem para valores semelhantes apontem, na prática, para o mesmo valor.

O programa:

```
x = 10.0
print(x)
x = 20.0
print(x)
```

Dá o resultado:

```
10.0
20.0
```

Quando fazemos uma atribuição também estamos, dinamicamente, determinando o tipo da variável `a`. Nesse caso, o tipo **float**. Podemos perguntar o tipo de qualquer variável usando a função **type(x)**.

O programa:

```
a = 10.0
```

¹ mais precisamente, ao espaço em memória que guarda esse valor. Isso vai fazer diferença no futuro

² Algumas linguagens de programação também possuem constantes, que tem um valor garantidamente fixo do início ao fim do programa

```
print(type(a))
```

Dá o resultado:

```
<type 'float'>
```

Podemos trocar o valor de *a*, e consequentemente seu tipo³. Por exemplo, o programa:

```
a = 10.0
print(type(a))
a = 10
print(type(a))
```

Dá o resultado:

```
<type 'float'>
<type 'int'>
```

O comando de atribuição é certamente o mais usado em programas. Sua lógica é a seguinte: atribua a variável a esquerda o valor da expressão a direita.

É importante notar que essa atribuição é imediata. Quando dizemos *a = 10*b* estamos dizendo que *a*, a partir desse momento, assume o valor de 10 vezes o valor de *b* nesse momento. Se trocarmos o valor de *b*, o valor de *a* permanece inalterado, porque a expressão *10*b* foi calculada e um valor foi gerado.

Chamamos a atenção, então, que o operador **=** **não** tem o significado de igual, mas sim de “atribuir a”.⁴

5.1 Atribuição Múltipla

No lado direito do operador **=** podemos colocar qualquer coisa que retorne um ou mais valores. Isso porque o operador é capaz de fazer a atribuição a várias variáveis ao mesmo tempo, como em⁵:

```
a = 10.0
b = 20.0
c, d = a, b
print(a, b, c, d)
```

Dá o resultado:

```
10.0 20.0 10.0 20.0
```

Isso é principalmente importante quando queremos trocar o resultado de duas variáveis, porque a avaliação das expressões a direita acontece antes das atribuições. Assim, todas as expressões são calculadas e então os valores são atribuídos as variáveis.

³ Lembramos que os tipos de Python são fortes e dinâmicos, isso significa que podemos trocar o tipo de uma variável ao longo de um programa

⁴ Isso não é bem como estamos acostumados na matemática e algumas linguagens, para deixar isso bem claro, usam o operador **:=**

⁵ Na verdade, uma atribuição múltipla usa o conceito de tuplas. Se o valor da expressão for uma tupla, então ela pode “desconstruir” a tupla e associar os valores individuais dos itens de tupla as variáveis nomeadas no lado esquerdo

```
a = 10.0
b = 20.0
b, a = a, b
print(a,b)
```

Dá o resultado:

```
20.0 10.0
```

Se você quisesse fazer a troca dos valores de `a` e `b` sem usar a atribuição múltipla, teria que guardar o valor de uma em outra variável⁶, para evitar que as duas ficassem com o mesmo valor.

⁶ Essa variável temporária é conhecida como *buffer*

```
a = 10
b = 20
c = 5
d = 7
# tentando trocar a e b de forma errada
a = b
b = a
# trocando c e d de usando um buffer
e = d
d = c
c = e
print(a,b,c,d,e)
```

Dá o resultado:

```
20 20 7 5 7
```

5.2 Valores e Ids

Cada valor possui um endereço de memória. Esse valor pode ser obtido com a função `id`. Veja a seguinte execução dessa função para diferentes variáveis e valores⁷.

⁷ Tente repetir em seu computador e verifique os resultados

```
1 >>> id(10.0)
2 2625655533688
3 >>> a=10
4 >>> id(a)
5 1998909488
6 >>> b=10
7 >>> c=10.0
8 >>> id(b)
9 1998909488
10 >>> id(c)
11 2625655533688
```

Nas linhas 1 e 2 podemos ver que mesmo valores `float` possuem um endereço. Bem mais trade, na linha 8 associamos o `float`

10.0 ao nome `c`. Mais tarde ainda, nas linhas 10 e 11 vemos que o endereço de `c` é exatamente o endereço de 10.0.

Vemos o mesmo acontecer com `a` e `b`, mesmo que nunca tenhamos dito que era iguais, eles apontam para o mesmo valor, a mesma constante, logo tem o mesmo endereço de memória.

6

Introdução a Funções

Já vimos algumas funções que são fornecidas incorporadas a linguagem Python. Agora veremos como usar outras funções e também como criar nossas próprias funções.

Estamos acostumados a usar funções na Matemática. Seno, cosseno, fatorial, raiz quadrada são funções aplicadas a números. As funções matemáticas normalmente exigem argumentos, como em \sqrt{x} .

6.0.1 Conceitos Sobre funções

Uma função é um subprograma que é chamado, no programa principal ou em outra função, para:

- fazer uma parte do programa principal que podemos isolar conceitualmente
- fazer uma parte do programa principal que é repetida em várias posições
- calcular um ou mais valores de forma isolada e repetitiva

Usamos muitas funções quando estamos resolvendo problemas reais. Por exemplo, a função que calcula o seno de um ângulo. Nós usamos ela em vários problemas trigonométricos sem nos preocupar como ela é feita. Muitas pessoas, com o tempo, até mesmo usam a função perfeitamente em problemas específicos sem se preocupar como uma calculadora ou programa de computador faz para calculá-la¹.

6.0.2 Coesão

Coesão é a característica de um pedaço de código fazer uma tarefa, com pouca interação com outros módulos. Normalmente queremos que nossas funções sejam extremamente coesas, isto é, tenham um só objetivo.

Os tipos de coesão, do qual desejamos sempre a mais alta, são [Pressman, 2010]:

- **Funcional** (a mais alta), quando o componente faz uma computação e retorna um resultado

¹ Na verdade, normalmente os programas calculam uma aproximação baseada em uma série, como a série de Taylor, que para seno é $\sin(x) \approx \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$

- **de Camadas**, quando módulos de nível mais alto usam módulos de nível mais baixo em uma arquitetura em camadas.
- **de comunicação**, onde operações que acessam o mesmo dado são definidos em um módulo

A coesão representa a força interna de um módulo.

6.0.3 Acomplamento

Acoplamento é a forma e o grau de interdependência entre partes de software. Basicamente, é a característica de um pedaço de código afetar outros pedaços. Normalmente queremos que nossas funções tenham baixíssimo, isto é, nenhum acoplamento com outras partes do código².

O seguintes tipos de acoplamento podem ser identificados[Pressman, 2010], sendo que quando mais baixo, melhor.

² O lema do acoplamento é “o que acontece em Vegas permanece em Vegas”

- **por conteúdo (altíssimo)**, onde um módulo modifica ou usa os dados de outro módulo.
- **global ou comum**, onde dois módulos estão acoplados pelo compartilhamento do mesmo dado global
- **por controle**, onde um módulo controla o fluxo de outro, passando informação sobre o que fazer, como uma *flag*.
- **por stamp**, onde um módulo é usado como tipo da operação de outro módulo.
- **por dados**, onde módulos compartilham muitos dados através de parâmetros.
- **por chamada**, onde um módulo chama simplesmente outro módulo, forma bastante comum.
- **por tipo**, onde um módulo usa um tipo de dados definido em outro.
- **por inclusão**, onde um módulo inclui um pacote que possui outro.
- **externa**, onde um módulo usa componentes de uma infraestrutura, como um sistema de arquivos.
- **por mensagens** (baixíssimo), a comunicação entre módulos é feita por passagem de mensagens, mas nesse caso um módulo não sabe para quem envia, o outro não sabe para quem responde.
- **nenhum**, onde os módulos não se comunicam um com o outro.

O acoplamento representa a independência de um módulo, seu isolamento de outros.

7

Usando Funções e Módulos

Nós já vimos que podemos usar funções em linguagens de programação.

Uma função, para o programador, é normalmente uma “caixa preta” que realiza algum cálculo ou processamento.

Funções possuem um nome, uma quantidade determinada ou não de argumentos ou parâmetros e retornam, ou não, um valor. Algumas funções alteram seus argumentos, outras não, de acordo com propriedades que veremos mais adiante.

Uma função bastante simples é aquela que calcula o valor absoluto de um `int` ou `float`.

```
>> abs(-133335.12)
133335.12
```

`abs(x)` é uma função que só aceita um argumento. Se você passar nenhum argumento ou mais de um argumento, você terá um erro.

Algumas funções podem ter mais de um argumento. Por exemplo, a função `pow(x, y)`¹ calcula x^y . Todos os argumentos devem ser passados na ordem certa.

Outra função interessante calcula o máximo de qualquer quantidade de números, `max(x, y, z, ...)`, ela aceita uma quantidade indefinida, maior que um, de argumentos.

```
>> max(2, 4.0, 3, 456.7)
456.7
```

Normalmente os argumentos devem seguir uma ordem específica e são obrigatórios. Python porém permite outros tipos de argumentos: argumentos com default e argumentos com nome.

7.1 Argumentos com default

Um argumento com default não precisa ser preenchido, e no caso ele assume o valor default. Argumentos default aparecem sempre

¹ Na verdade, a função `pow` tem um terceiro argumento `t`, opcional, e que permite calcular $x^y \% z$

depois dos argumentos obrigatórios em uma função.

Por exemplo, a função `complex(real[, imag])`² aceita um ou dois parâmetros. Se ela receber um parâmetro só, supõe que seja um valor real que deve ser convertido para um valor complexo que só tem a parte real (isso é, `complex(5)` retorna `5+0j`). Se receber os dois valores, considera a parte real e a imaginária. A função `complex` assume que `imag` tem o valor default de zero.

² Essa notação usando conchetes será usada sempre para indicar que um parâmetro é opcional. Essa é a notação oficial do site de Python, porém ela **não** é usada enquanto programamos.

```
>> complex(5)
```

```
(5+0j)
```

```
>> complex(5,6)
```

```
(5+6j)
```

7.2 Argumentos com nomes

Todos os argumentos de uma função tem nome. Isso faz com que seja usar os argumentos pelos nomes, e não pela ordem. Nesse caso, podemos usar os parâmetros fora de ordem. Isso é feito como no exemplo a seguir:

```
>> complex(imag=5, real=6)
```

```
(6+5j)
```

Isso é interessante para funções que tem muitos, ou todos, valores com default, pois aí podemos selecionar mudar o valor apenas daqueles que queremos. Mais tarde veremos que a função `print` possui muitas opções que mudam a forma como a impressão é feita.

Existem 68 funções que vem incorporadas na linguagem. Elas são descritas em <https://docs.python.org/3/library/functions.html>. Mais Python fornece talvez milhares de funções, alguns junto com a linguagem, outros fornecidos por terceiros. Para usarmos essas funções temos que usar módulos.

7.3 *args e **kwargs

Muitas descrições de funções em Python usam os termos `*args` e `**kwargs`.

`*args` significa uma quantidade ilimitada de argumentos, sem nome. `**kwargs` significa uma quantidade ilimitada de argumentos nomeados. O uso dessa estrutura permite que uma função não saiba que argumentos ela atende, mas que possa passar todos os argumentos recebidos para outras funções.

Por exemplo, `print` é uma função que usa `*args` e dessa forma pode aceitar uma infinidade de valores para imprimir.

É mais difícil achar funções que usam `**kwargs` nos pacotes mais simples, porém no pacote gráfico `matplotlib`³ várias funções mais complexas recebem argumentos com nome não definidos e simplesmente passam esses argumentos para sub funções.

³ <http://matplotlib.org/>

7.4 O que são Módulos

MÓDULOS SÃO ARQUIVOS EM PYTHON QUE TEM, DENTRO DELES, VÁRIAS FUNÇÕES DEFINIDAS. Na definição oficial, “módulos são objetos que servem como uma unidade organizacional do código Python”. Na prática, estendem a linguagem com mais funções do que aquelas que já vem incorporadas.

Um módulo bem feito agrupa um conjunto de funções que tem algum objetivo comum. A maioria das linguagens de programação possui esse mecanismo, conhecido como biblioteca padrão, ou pelo termo em inglês *standard library*, pois é interessante separar o núcleo da linguagem de partes que não são interessantes para todos. Assim, quando você quer fazer alguma coisa específica, incorpora o módulo a linguagem por meio de algum comando.

Alguns módulos padrão Python aparecem na Tabela 7.1.

Módulo	Utilidade
<code>math</code>	Funções matemáticas
<code>cmath</code>	Funções matemáticas com complexos
<code>array</code>	Funções eficiente para arrays
<code>random</code>	Funções que geram números aleatórios
<code>csv</code>	Funções para leitura e escrita de arquivos csv
<code>email</code>	Funções para tratar emails

Tabela 7.1: Alguns Módulos de Python

Existem muito mais módulos fornecidos com uma instalação Python, que podem ser vistos em <https://docs.python.org/3.4/library/index.html>.

Módulos também definem *namespaces*, que permitem que vários módulos tenham funções com o mesmo nome. Por exemplo, ambos os módulos `math` e `complex` possuem uma função `sqrt(x)`. Você pode usar as duas usando `math.sqrt(x)` e `complexath.sqrt(x)`, usando o nome do módulo como qualificador. As duas funções existem em espaços de nome, *namespaces*, diferentes.

Em Python, o comando que permite incorporar um módulo a um programa, ou a um ambiente em execução é o comando `import`.

7.5 Usando Módulos

Para usar um módulo são possíveis alguns comandos diferentes.

A primeira forma do comando importa o módulo para o ambiente de execução, mas exige que chamemos a função pelo seu nome completo, isso é, utilizando o nome do módulo.

```
import math
math.sqrt(12)
```

A segunda forma permite renomear o módulo, para facilitar a tarefa de digitação ou usar dois módulos com mesmo nome, mas ainda exige que indiquemos o *namespace* da função

```
import math as m
m.sqrt(12)
```

A terceira forma permite import todas as funções do módulo, eliminando a necessidade de qualificar com o seu nome.

```
from math import *
sqrt(12)
```

E a quarta forma, a que devemos dar preferência nos programas, permite importar uma, ou algumas, funções de um mesmo modo, mas não necessariamente todas.

```
from math import sqrt
sqrt(12)
```

Finalmente, para quando precisamos importar funções com mesmo nome de módulos diferentes, é possível funções dos módulos trocando também os seus nomes, como em:

```
from math import sqrt as s
s(12)
```

A seguir, um programa que calcula algumas funções sobre alguns números:

```
from math import sqrt, log10
from cmath import sqrt as csqrt
from random import *
import time
print(sqrt(10), log10(1000)) # uso direto
print(csqrt(10+10j)) #uso direto com nome trocado
print(randint(1,10)) # uso direto
print(time.clock()) # uso com nome qualificado
```

Com resultado:

```
3.16227766017 3.0
(3.4743442276+1.43912049943j)
1
0.02
```

7.6 Problemas com Funções Matemáticas e Similares

Exercício 10. Álvaro e Bianca estão jogando gamão, mas perderam os dois dados que caracterizam o jogo. Que código simples em Python substitui o dado? Devem digitar no interpretador uma vez:

```
from random import randint
```

Isso é necessário para importar o módulo de valores aleatórios. E sempre que precisarem dos dois dados, devem executar:

```
randint(1,6),randing(1,6) .
```

Isso resultará em dois números aleatórios seguidos⁴.

Exercício 11. A função `time()` do módulo `time` retorna o tempo em segundos desde Epoch⁵. Dois programadores, Abel e Bruno, apostaram que podiam subir e descer as escadas de seu prédio mais rapidamente. Para que um não interfira no outro, decidiram que um iria primeiro, teria o tempo contado e o outro iria depois, tendo o tempo contado também. Para contar o tempo eles teriam que executar (com um toque na tecla Enter um pequeno pedaço de código que fizesse a conta. Como Abel e Bruno podem usar a função `time()` para isso?

Primeiro, devem importar a função.

```
from time import time
```

Ao iniciar a contagem de tempo, o primeiro corredor executará a linha:

```
inicial = time()
```

Enquanto isso o segundo corredor, que está esperando, prepara a próxima linha de código, que o primeiro corredor executará quando chegar.

```
fim1 = time()
```

Depois, o mesmo é repetido para o segundo corredor, porém as variáveis podem se chamar `inicia2` e `fim2`. Ao acabar a corrida eles podem calcular os tempos com as linhas

```
fim1 - inicial
```

```
fim2 - inicia2
```

Aquele com menor tempo é o vencedor.

7.7 Pacotes

Pacotes são módulos complexos construídos com vários arquivos organizados de forma hierárquica.

7.7.1 Usando pacotes de outros programadores

Vários pacotes e módulos de Python independentes podem ser encontrados no *Python Package Index - PyPI* em <https://pypi.python.org/pypi>. Existem ainda outros pacotes que podem ser obtidos de pessoas ou empresas específicas e não estão listados.

Em fevereiro de 2016 existiam 75108 pacotes listados em PyPI. Essa é uma das maiores forças de Python, a existência de milhares de pacotes que fazem tarefas que, de outro modo, teríamos que programar.

⁴ Em gamão não se usa a soma, mas os dados em separado

⁵ Epoch é momento inicial onde computadores contam o tempo. No Microsoft Windows 10 é 0:00 1/1/1601, no Unix é 0:00 de 1/1/1970

7.8 Métodos

Uma classe especial de funções são os métodos⁶. Métodos são funções criadas dentro de classes (os tipos de Python) e são invocados com a notação ".".

⁶ Métodos podem ser melhor entendidos sob o conceito de Classe

Quando discutimos tipos de dados, afirmamos que os tipos incluem as representações e o comportamento dos dados. O comportamento é caracterizado por funções especiais, que se são os métodos.

Sempre que temos acesso a um tipo, como `str` várias métodos, diretamente associados ao tipo, ficam disponíveis. Eles podem ser usados de duas formas.

A primeira forma é usá-los como uma função comum, usando a notação `tipo.método`. Por exemplo, para encontrar a string "alvo" dentro de uma string na variável `s`, escrevemos `str.find(s, "alvo")`.

É mais comum⁷ fazer as chamadas como método. A notação então é `valor.método` e não é necessário fazer referência novamente ao valor⁸, como em `s.find("alvo")`

⁷ Oi mais pythônico

Ou seja, **todo método tem um parâmetro implícito**, inicial, que é o objeto sobre o qual ele foi chamado, pelo operador ".".

⁸ Pode ser uma variável ou uma constante de valor

Como método:

```
>>> "sajdaskljd".find("jd")
```

Dá o resultado:

```
2
```

Como função da classe:

```
>>> str.find("sajdaskljd", "jd")
```

Dá o resultado:

```
2
```

7.9 Métodos para Strings

A seguir uma lista de algumas funções para strings.

- `str.upper(s)`, retorna uma string passando os caracteres de `s` para letras maiúsculas
- `str.lower(s)`, retorna uma string passando os caracteres de `s` para letras minúsculas
- `str.count(s1, s2, inicio, fim)`, retorna quantas vezes `s2` aparece em `s1`, entre as posições `inicio` e `fim-1`.
- `str.index(s1, s2, inicio, fim)`, retorna a posição da primeira aparição de `s2` em `s1` entre `inicio` e `fim-1`. Causa um erro se `s2` não está em `s1`

7.10 Funções que retornam tuplas ou vários valores

Algumas funções retornam vários valores. Na prática elas retornam tuplas. Assim, uma função que retorna vários valores pode ter esses valores associados a uma variável ou a mais de uma variável (o mesmo número de valores que retorna).

A função `divmod(x,y)`, por exemplo, pode ser usada de duas formas.

```
t = divmod(10,3)
print(t)
q, r = divmod(10,3)
print(q,r)

(3, 1)
3 1
```

7.11 Programas com String e seus Métodos

Exercício 12. Uma agenda telefônica contém a string "Deofantini,Carlos", com o sobrenome antes do nome. Transforme essa string, colocando-a em uma variável `novo_nome`, para uma string onde o nome está na frente do sobrenome, separados por espaço.

```
1 nome_original = "Deofantini,Carlos"
2 posicao = str.index(nome_original, ",")
3 sobrenome = nome_original[:posicao]
4 nome = nome_original[posicao+1:]
5 novo_nome = nome + "_" + sobrenome
```

Essa solução tem o seguinte objetivo, em cada linha:

1. Coloca a string em uma variável, assim ficará mais fácil lidar com ela sem repeti-la várias vezes e correr o risco de errar uma digitação. Lembre que Python verificará se sua variável existe, mas não pode verificar se você está escrevendo uma string do mesmo jeito todas as vezes.
2. Calcula a posição da vírgula na string
3. Calcula o nome, pegando a string até a posição.
4. Calcula o sobrenome, pulando a vírgula
5. Concatena nome e sobrenome na forma usual

Exercício 13. Aldo e Beatriz contaram quantas vezes a letra "i" aparece na palavra "inconstitucionalicimamente", mas chegaram a resultados diferentes. Alguém não prestou atenção? Ajude os dois, fazendo essa conta sem nenhuma sobre de dúvidas.

```
str.count("inconstitucionalicimamente","i")
```


8

Criando Funções

ALÉM DAS FUNÇÕES PRÉ-DEFINIDAS NA LINGUAGEM, NORMALMENTE QUEREMOS DEFINIR TAMBÉM AS NOSSAS FUNÇÕES. Isso acontece por vários motivos, mas o principal é a estruturação do código de forma a entendermos melhor.

Assim, se separamos o código em pedaços que podem ser bem entendidos, então fica mais fácil não só implementá-lo corretamente, como também corrigi-lo no futuro.

Para criar uma função usamos o comando `def`. Toda função retorna um valor e isso acontece com o comando `return`. Quando o código de uma função encontra o comando `return` ele termina imediatamente, não interessando se existem outras linhas de código naquela função. Em Python, essa característica é usada fortemente na programação.¹

O programa a seguir apresenta uma função muito simples que retorna o número 1.

```
def simples():  
    return 1
```

Funções que não recebem nenhum parâmetro são úteis em alguns casos, mas é mais normal que elas recebam argumentos que permitam que seu comportamento seja alterado em função do valor dos argumentos.

8.1 Identação e blocos em Python

Nesse ponto temos que parar e entender um pouco mais do porque a palavra `return` estar indentada em relação a palavra `def` no programa anterior.

Em Python a indentação é usada para marcar grupos de linhas de código que estão subordinadas a uma instrução de alguma forma. Um grupo de linhas de código sempre pertence a algum nível de código. Enquanto a indentação não muda, o grupo de linhas de código é o mesmo.

O primeiro nível de indentação é a primeira coluna da linha. Se você colocar um espaço em um arquivo Python no lugar da primeira coluna ao começar um programa receberá o erro `unexpected`

¹ Se você não retornar nada, ou seja, se sua função terminar sem nunca passar por um comando de `return`, o interpretador Python considera que foi retornado o valor `None`.

indent.

Tudo que estiver indentado imediatamente a seguir da instrução `def` forma a definição da função. Essas linhas passam a pertencer a definição da função. Essa definição pode ser muito longa, ter linhas vazias ou comentários no meio. Porém, assim que a indentação voltar a posição original, a definição da função acabou.

No exemplo abaixo explicamos que linhas pertencem a definição da função.

```
def funcao_longa(x,y,z):
    a = 2 * x
    b = 3 * y
    # comentários podem estar indentados
# comentários podem não estar indentados
    c = 4 * z # continua na função

    return a*b*c # continua na função, linhas em branco ok

d = funcao_longa(1,2,3) # voltou ao mesmo nível, não é mais função
print(d)
```

8.2 Voltando as Funções

A função a seguir recebe um argumento, `x`, e retorna o seu triplo.

```
def triplo(x):
    return 3*x
```

E a função a seguir retorna o produto de dois números.

```
def produto(x,y):
    return x,y
```

Claro que funções tão simples tem pouca utilidade. Uma função muito mais útil recebe o seu peso em kg, sua altura em metros e calcula o seu índice de massa corporal (IMC):

```
def imc(peso,altura):
    return peso / altura**2
```

Agora sim temos uma função bastante útil, que serve para calcular o imc e pode ser usada em vários programas. Ela também faz a conta e esconde o processo da gente, ou seja, não precisamos mais saber como o imc é calculado, apenas que ele precisa de peso e altura.

Mas, e se não soubermos como usar o peso e a altura? Seriam em grama, quilograma, metro, centímetros? Bem, a linguagem propriamente dita não nos permite tratar isso. Não há como perguntar se um número do tipo `float` representa metros ou quilômetros. Só nos resta orientar, de alguma forma, o programador que vai usar a função.

Nós podemos e devemos usar comentários especiais em nossas funções que podem ser vistos pelo programador com a função `help`.

Esses comentários aparecem entre aspas logo após a definição da função.

O programa ficaria assim:

```
def imc(peso,altura):
    "calcula_imc_a_partir_do_peso_em_kg_e_altura_em_m"
    return peso / altura**2
```

Obviamente, sendo parte de Python, dentro de funções podemos chamar outras funções, o que é realmente muito usado para deixar os programas mais claros. Veja o exercício 15

8.3 Funções e Subfunções

Dentro de uma função podemos definir funções. Isso faz com que a função mais interna só possa ser usada dentro do corpo da função mais externa.

A função `fext(x)` a seguir calcula $x^x \times x^x$.

```
def fexter(x):
    def fint(y):
        return y**y
    return fint(x)*fint(x)
```

8.4 Argumentos Opcionais ou Default

É possível colocar argumentos opcionais em uma função. Esses argumentos devem vier obrigatoriamente após os argumentos obrigatórios e devem conter um valor default. Isso é feito como no exemplo a seguir, onde `base` é um argumento *default*.

```
def peso(massa,g=9.806):
    "peso_em_N,_dado_a_massa_em_kg_e_a_aceleracao_da_gravidade_em_m/s^2"
    return massa*g
```

8.5 Quantidade indefinida de argumentos

É possível passar uma quantidade indefinida de argumentos usando uma notação especial, com o asterisco `*` antes do nome de um argumento que será considerado uma lista.

```
def f(*x):
    soma = 0
    for i in x:
        soma += i
    return soma

print f(1,2,3,4,5,6)
```

Resulta em:

21

Há alguma interação entre os argumentos obrigatórios, o argumento de múltiplos valores e os argumentos default. Os argumentos obrigatórios, simples sem default sempre devem vir antes.

8.6 Argumentos nomeados indefinidos - ***kwargs*

O uso do símbolo `**` permite usar argumentos nomeados indefinidos (e sem *default*). Todos os argumentos nomeados são colocados em um dicionário onde as chaves são os nomes e os valores os valores passados.

```
def f(**d):
    soma = 0
    for k in d:
        print k,d[k]
f(cor="vermelho", tamanho=10, peso=4)
```

Resulta em:

```
cor vermelho
tamanho 10
peso 4
```

Os argumentos nomeados indefinidos devem aparecer depois dos argumentos indefinidos não nomeados.

8.7 Escopo das Variáveis

Quando declaramos uma função abrimos um novo espaço de nomes para nossas variáveis. Chamamos isso de um novo **escopo**. Isso significa que as variáveis usadas nas funções são, normalmente, novas variáveis e não são as mesmas usadas no programa principal ou em um outro escopo qualquer.

As seguintes regras são válidas:

- Se a variável é declarada como um argumento da função, ela é uma variável nova. Mesmo que tenha o mesmo nome de outra variável em qualquer outro lugar do programa, ela é outra variável.
- Se a variável recebe um valor dentro do corpo da função ela é uma variável nova.
- Se a variável é declarada `global`, então ela não é nova, e sim é uma variável do módulo.

Na prática, quando encontra uma variável, Python faz os seguintes passos:

1. Procura a variável no *namespace* local, por exemplo no escopo da função
2. (caso não seja local) procura a variável no *namespace* global, ou seja, no âmbito mais externos de todo o módulo, sem entrar nos *namespaces* internos.

3. (caso ainda não tenha encontrado) procura a variável no âmbito de todo o programa

Por exemplo, no programa abaixo:

```
a = 1
b = 3
e = 4
```

```
def funcao_longa(x,y,z):
    global c
    a = 2 * x
    b = 3 * y
    # comentários podem estar indentados
# comentários podem não estar indentados
    c = e * z # continua na função

    return a*b*c # continua na função, linhas em branco ok

def funcao_complexa(x,y,z):
    c = 200
    def funcao_interna(x,y):
        global c,a
        c = 100
        return x+y+c+a
    g = funcao_interna(x,x)
    return x+y+z+g

d = funcao_longa(1,2,3)
f = funcao_complexa(9,10,11)
print(a,b,c,d,e,f)
```

Acontece o seguinte, passo a passo:

1. a linha `a = 1` é executada, `a` é criado no escopo global, com valor 1
2. a linha `b = 3` é executada, `b` é criado no escopo global, com valor 3
3. a linha `e = 4` é executada, `e` é criado no escopo global, com valor 4
4. a função `função_longa` é analisada, não é encontrado erro, e ela é salva no escopo global, porém não é executada, pois é só uma definição.
5. a função `função_complexa` passa pelo mesmo processo, porém dentro dela aparece uma nova definição de função, a `funcao_interna`, e só pode ser vista dentro de `funcao_complexa`.
6. `d` aparece, mas para dar um valor a `d` precisamos antes calcular o valor de `função_longa(1,2,3)`.

- (a) A função é chamada e, na ordem, x, y, z recebem os valores 1, 2, 3.
- (b) c é declarado global, isso significa que a variável c é válida escopo global
- (c) a e b aparecem, mas nesse caso são privativas desse *namespace*, logo, seus valores não afetam os valores dos a, b originais
- (d) um novo valor para c é calculado, que vai afetar a variável global.
- (e) A função retorna um valor.

7. d recebe o valor calculado

8. novamente, precisamos calcular o valor de f , para isso temos que entrar em `funcao_complexa`.

- (a) x, y, z recebem o valor de 2, 2, 2.
- (b) c recebe o valor de 200, mas esse é um novo c .
- (c) g é calculado, para isso é necessário chamar `funcao_interna`
 - i. x, y recebem o valor 10 e 10
 - ii. c, a são declaradas globais
 - iii. c é alterado
 - iv. A função retorna um valor.

9. As variáveis são impressas

Esse programa tem o resultado:

```
1 3 100 144 4 149
```

Vamos analisar esse resultado. As variáveis a e b recebem um valor no programa. Outras variáveis como mesmo nome, mas em um *namespace* diferente recebem outros valores, mas isso não afeta o valor delas.

Já a variável c só é criada quando `funcao_longa` é chamada. Ela não existe no contexto global até esse momento, e nunca recebeu um valor ou foi criada no código do contexto global. Porém, dentro de uma função, ela foi criada no contexto global. Ela não foi criada na sua declaração, mas sim quando recebe o valor e já foi declarada como global. Novamente ela é declarada global em `funcao_complexa` e `funcao_interna`, então ela muda de valor.

8.8 Mais no dinamismo da linguagem

O programa seguir responde com um erro. Isso acontece porque a variável a não existe no contexto global quando `print` é chamada.

Já o programa a seguir imprime o número 1

```
def f():
    global a
    a = 1
f()
print a
```

Por que? Porque ao executar a variável `a` é criada no contexto global, dentro da função `f()`

Esse dinamismo da linguagem é uma característica muito importante.²

² Outras linguagens, como C, possuem escopo estático. Algo como isso não funcionaria, pois ao compilar diria que `a` não existe

8.9 Anotações

Curiosamente, Python permite você anotar os argumentos, porém esses argumentos **nada significam** e só ganham significado se estiver sendo usada uma biblioteca específica que os use.

O programa a seguir mostra tipos diferentes de argumentos, e nenhum deles garante alguma coisa, funcionando na prática como comentários.

```
def f(x: float , y : int = 0 , z : "nada_feito") -> float:
    pass
```

As anotações são feitas com o símbolo `:` para argumentos ou o símbolo `->` para a função. Cada anotação deve ser uma expressão. As anotações aparecem antes do valor *default*, por isso `y : int = 0`, onde `y` terá o valor *default* de 0, e a anotação `int` nada significa para o interpretador, podendo ser usada como indicador apenas para os seres humanos.

É interessante que Python tenha adotado essas anotações, pois elas levam um programador mais experiente em outras linguagens a pensar que `x` tem que ser `float`, mas nada mais longe da verdade.

Essa mudança foi feita na PEC 3107.

8.10 Decoradores

Decoradores são comandos precedidos do símbolo `@` que aparecem antes da definição de funções. Eles têm significados específicos que devem ser vistos um a um.

Decoradores aparecem na PEC 318

8.11 Problemas com Funções

Exercício 14. Faça uma função `traduz(s)` que receba um nome na forma tradicional de agendas americanas "sobrenome, nome" e devolva na forma tradicional de agendas brasileiras "nome sobrenome". Note que após a vírgula há sempre um espaço. Use a vírgula como delimitador na forma americana³.

³ Esse problema generaliza o exercício 12

```
1 def traduz(s):
2     "Recebe_string_na_forma_sobrenome,_nome_e_devolve_na_forma_nome_sobrenome"
3     posicao = str.index(s, ",")
4     nome = s[posicao+2:] # pula vírgula e espaço
5     sobrenome = s[:posicao]
6     return nome+"_"+sobrenome
```

A linha 3 pode usar também a notação de método.

```

1 def traduz(s):
2     "Recebe_string_na_forma_sobrenome,_nome_e_devolve_na_forma_nome_sobrenome"
3     posicao = s.index(",")
4     nome = s[posicao+2:] # pula vírgula e espaço
5     sobrenome = s[:posicao]
6     return nome+"_"+sobrenome

```

Exercício 15. Faça uma função que receba duas strings indicando o tempo em horas, minutos e segundos, segundo a notação "HH:MM:SS" e retorne a diferença entre eles também na mesma notação. Use funções auxiliares para evitar repetir código.

```

1 def horas(s):
2     return int(s[:2])
3
4 def minutos(s):
5     return int(s[3:5])
6
7 def segundos(s):
8     return int(s[6:])
9
10 def tempo(h,m,s):
11     return h*3600+m*60+s
12
13 def converte(s):
14     return tempo(horas(s),minutos(s),segundos(s))
15
16 def diferenca(s1,s2):
17     dif = converte(s2)-converte(s1)
18     horas, resto = divmod(dif,3600)
19     minutos, segundos = divmod(resto,60)
20     return str(horas)+":"+str(minutos)+":"+str(segundos)

```

As primeiras funções servem para retirar de uma string um inteiro já significando as horas, minutos e segundos. Elas usam nosso conhecimento da posição exata desses números na string definida. Já na linha 10 temos uma função que converte uma medida em horas, minutos e segundos, agora em inteiros e separados, em segundos. A função converte serve para aplicar, em uma só chamada esses dois processos, de uma string a um número de segundos.

Finalmente a função diferenca faz as conversões e calcula a diferença, que é depois convertida na string. Atenção ao uso da função `divmod`, que retorna simultaneamente o quociente e o resto de uma divisão inteira, que são colocados em duas variável por meio da dupla atribuição.

9

Entrada e Saída

Um programa de computador tem que receber e fornecer informações. Esse capítulo trata das formas mais simples como isso pode ser feito em Python.

9.1 Imprimindo Resultados

Um programa não será útil se não fornecer alguma informação a quem o invoca. Para isso, os programas devem normalmente imprimir, no vídeo, no arquivo ou em outro dispositivo de saída, alguma coisa. Não seria ousado dizer que não é possível conceber um programa de computador útil que não tivesse alguma forma de saída, não necessariamente legível por uma pessoa.

Para imprimir no vídeo, na sua forma mais simples, Python usa a função `print`. Essa função aceita uma quantidade ilimitada de argumentos, que são impressos um após o outro. Ela também aceita alguns argumentos especiais que explicam como deve ser uma expressão.

Alguns programas exemplos nesse livro já usaram o `print`. Para usá-lo, basta usar como argumento todas as variáveis que você deseja imprimir¹

A função `print` é descrita pelo site de Python como:

```
print(*objects, sep='_ ', end='\n',  
      file=sys.stdout, flush=False)}
```

Isso significa que ela responde a um número indefinido de objetos e que 4 argumentos podem ser definidos (e já vem com o valor default). A Tabela 9.1 mostra esses argumentos.

Argumento	Utilidade	Default
<code>sep</code>	separador usado entre cada objeto impresso	espaço
<code>end</code>	separador usado entre cada chamada ao <code>print</code>	fim de linha
<code>file</code>	arquivo de saída a ser utilizada pelo <code>print</code>	saída padrão
<code>flush</code>	se deve esvaziar o buffer imediatamente	não fazer

Alterar o parâmetro `sep` não é muito comum, mas é possível. Algumas vezes trocamos o parâmetro para a string vazia `,` para que nada separe o que estamos imprimindo.

O parâmetro `end` é dos mais alterados, pois usamos muito `sep=""` para dizer que não queremos pular linha após imprimir.

¹ Isso funcionará para qualquer tipo que tenha a função `str`

Tabela 9.1: Opções do comando `print`

Os parâmetros `file` é usado quando queremos imprimir em um arquivo, ou em outra saída, mas não é tão comum². O parâmetro `flush` é ainda mais incomum e normalmente usado para ajustar algum comportamento indesejado³.

```
print("um_texto")
print("o_print_seguite_pula_linha")
print("mas_podemos_fazer_continuar_", end="")
print("onde_parou")
print("ou_po", "demos_tirar_o_separador", sep="")
print("ou_separar", "de", "outra", "forma", sep="+")
```

Dá o resultado:

```
um texto
o print seguinte pula linha
mas podemos fazer continuar onde parou
ou podemos tirar o separador
ou separar+de+outra+forma
```

9.2 Entrada de dados

A função `input` permite que um valor seja obtido do usuário. Ela imprime uma mensagem e espera que o usuário a responda.

Em Python 3, `input` sempre retorna uma string⁴.

Como ela retorna sempre uma string, para obter valores numéricos, ou outros, você terá que fazer algum processamento. Para os tipos numéricos é fácil, basta usar as funções de conversão.

```
booleano = bool(input("Entre com um booleano: "))
inteiro = int(input("Entre com um int: "))
flutuante = float(input("Entre com um float: "))
complexo = complex(input("Entre com um complex: "))
string = input("Entre com uma string: ")
print(booleano, inteiro, flutuante, complexo, string)
```

Dá o resultado (incluindo as entradas):

```
Entre com um booleano: True
Entre com um int: 12
Entre com um float: 23.0
Entre com um complex: 34+2j
Entre com uma string: cadeia de caracteres
True 12 23.0 (34+2j) cadeia de caracteres
```

9.3 Problemas com Entrada e Saída de Dados

Exercício 16. Faça um programa que pergunte dois números ao usuário e imprima a média dos dois números.

² Por exemplo, normalmente imprimimos erros em `sys.stderr`

³ Em dispositivos *bufferizados* os dados enviados não são salvos imediatamente, e sim ficam esperando para ser salvos quando alguma condição, como o fechamento do dispositivo, acontece. Isso pode permitir que dados cheguem ao dispositivos em uma ordem e sejam salvos em outra, ou que não sejam salvos em caso de erro

⁴ Em Python 2 `raw_input` sempre retorna uma string e `input` analisa qual o tipo de dado, retornando-o corretamente. Porém, para retornar uma string ela deve receber a string entre aspas (simples ou duplas), caso contrário retornará erro


```
n1 = float(input("Entre_com_o_primeiro_número:_"))
n2 = float(input("Entre_com_o_segundo_número:_"))
print("A_média_dos_números_é:_", (n1+n2)/2.0)
```

Exercício 17. André possui notas de 1,2,5,10,20,50 e 100 reais. Faça um programa que pergunte uma quantia e imprima a menor quantidade de notas de cada valor necessária para alcançar a quantia. Por exemplo, para R\$152,00 seriam necessárias 1 nota de R\$2,00, 1 nota de R\$50,00 e uma nota de R\$100,00. Use valores inteiros.

```
1 quantia = int(input("Entre_com_um_valor_(inteiro):_"))
2 notas_100 = quantia // 100
3 resto = quantia % 100
4 notas_50 = resto // 50
5 resto = resto % 50
6 notas_20 = resto // 20
7 resto = resto % 20
8 notas_10 = resto // 10
9 resto = resto % 10
10 notas_5 = resto // 5
11 resto = resto % 5
12 notas_2 = resto // 2
13 resto = resto % 2
14 notas_1 = resto
15 print("Notas_de_100:_",notas_100)
16 print("Notas_de_50:_",notas_50)
17 print("Notas_de_20:_",notas_20)
18 print("Notas_de_10:_",notas_10)
19 print("Notas_de_5:_",notas_5)
20 print("Notas_de_2:_",notas_2)
21 print("Notas_de_1:_",notas_1)
```

A solução se aproveita da capacidade de alterar o valor da variável `resto`, nas linhas 3,5,7,9,11 e 13. Não precisamos, claramente, dividir o último resto por 1.

Programas em Python

UM PROGRAMA EM PYTHON É SIMPLEMENTE UM ARQUIVO. A função de um programa de computador é processar informações. Por exemplo, dado 10 valores ele pode calcular a média desse valores. Para isso, é necessário que programas de computador leiam dados e escrevam dados. Eles podem fazer isso a partir de interação com o usuário, por meio do teclado e do monitor, ou por meio de arquivos, redes de computador, etc.

10.1 Programa Edita-Processa-Escreve

Um forma simples de escrever programas é não fazer a leitura de dados, colocando todos os dados dentro desse programa. Essa forma é comum no aprendizado e algumas vezes útil quando precisamos fazer coisas só uma vez.

Por exemplo, um programa que calcula a média de 3 números poderia ser escrito assim.

```
# lendo os valores
a = 10.0
b = 5.0
c = 4.0
# calculando a media
media = (a + b + c) / 3.0
# imprimindo o resultado
print("Resultado_", media)
```

Veja que cada vez que esses números mudarem, temos que mudar o programa. Algumas vezes, durante o desenvolvimento, parece que essa é uma solução mais fácil, pois evita que digitemos os dados a cada momento. A verdade é que a solução mais correta seria ler dados de um arquivo. Assim, podemos mudar tanto o arquivo de dados quanto o programa de forma independente.

Python, como é uma linguagem interpretada, se presta bastante a essa forma de programar, mesmo ela não sendo a ideal.

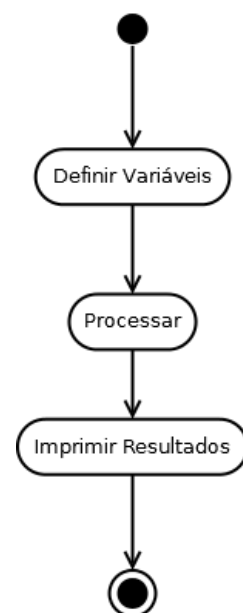


Figura 10.1: Programa do Tipo Define-Processa-Escreve

10.2 Programa Lê-Processa-Escreve

Um programa típico de computador tem 3 passos: ler todos os dados, processar os dados e escrever todos os resultados.

O programa a seguir, por exemplo, calcula a média de 3 números fornecidos pelo usuário, usando funções para obter e imprimir os números.

```
# lendo os valores
a = float(input("Entre_com_o_primeiro_valor:_"))
b = float(input("Entre_com_o_segundo_valor:_"))
c = float(input("Entre_com_o_terceiro_valor:_"))
# calculando a media
media = (a + b + c) / 3.0
# imprimindo o resultado
print("Resultado_",media)
```

Nesse programa 3 funções são usadas. A primeira função é `input`, que faz uma pergunta ao usuário e retorna um valor do tipo `str`. A segunda é a função `float`, usada aqui para transformar uma string no número que ela representa. Finalmente, a função `print` imprime resultados. Além disso, ele usa o operador de atribuição `=`.

Esse programa é melhor, ou mais genérico, que o primeiro tipo, pois não precisamos modificá-lo para obter novos resultados. Mesmo assim, precisamos executá-lo várias vezes.

Quando lê de arquivos, e não do teclado, esse programa é comumente chamado de programa em *batch*. No passado todos os programas rodavam dessa forma. Hoje, é comum na Engenharia ou em outras ciências, fazermos programa que rodam dessa forma, em busca de resultados seguidos de um projeto com parâmetros diferentes.

É interessante notar que apesar de usarmos esse paradigma, os dados não precisam ser realmente lido todos de uma vez. A questão importante é que os dados são lidos antes de serem processados e não há nenhuma decisão de continuar ou não baseada nos dados. Um programa que segue o mesmo paradigma é o seguinte:

```
a = float(input("Entre_com_a_primeira_notas:_"))
b = float(input("Entre_com_a_segunda_notas:_"))
media = (a + b) / 2.0
print("Média_das_duas_primeiras_provas_",media)
c = float(input("Entre_com_a_notas_da_terceira_prova:_"))
media = (a + b + c) / 3.0
print("Média_das_três_provas_",media)
```

10.3 Programa Lê-Processa-Escreve-Repete

Os programas acima precisavam ser executados para cada conjunto de dados, mas as vezes nossos dados são repetidos e não sabemos quantas repetições, ou precisamos de mais dados até um certo

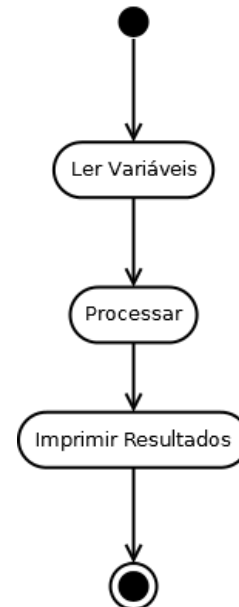


Figura 10.2: Programa do Tipo Lê-Processa-Escreve

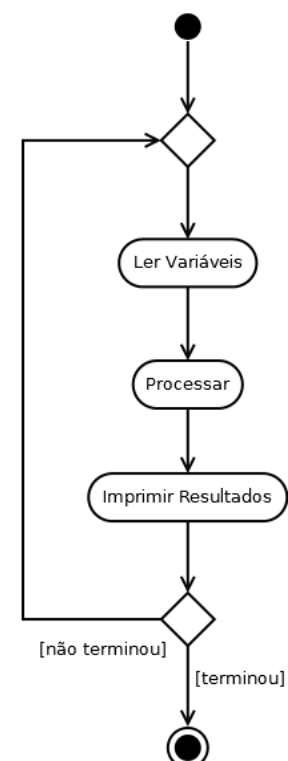


Figura 10.3: Programa do Tipo Lê-Processa-Escreve-Repete

ponto. Nesses casos a estratégia é colocar todo o programa dentro de um grande laço, que vai executar até que uma condição seja encontrada.

Para exemplificar esse tipo de programa temos que usar uma instrução `while`, ainda não apresentada.

```
while True:
    # lendo os valores
    a = float(input("Entre_com_o_primeiro_valor:_"))
    b = float(input("Entre_com_o_segundo_valor:_"))
    c = float(input("Entre_com_o_terceiro_valor:_"))
    # calculando a media
    if a==-1 and b==-1 and c==-1:
        break
    media = (a + b + c) / 3.0
    # imprimindo o resultado
    print("Resultado_",media)
```

Esse programa executa sem parar até que o usuário entre com `-1` para todos os valores pedidos. Nesse caso, ele para. Esse tipo de programa evita que tenhamos que reexecutá-lo várias vezes, com uma só execução todos os dados são processados.

11

Tomada de decisão - *if*

A tomada de decisão é uma das características típicas de algoritmo. Praticamente todas as linguagens de programação possuem uma instrução que permite decidir se uma ação vai ser feita em função de uma pergunta.

A instrução para isso em Python é a instrução `if`, que significa *se* em português. Ela apresenta uma forma básica:

```
if <expressão com valor verdade>:  
    <bloco>
```

Onde *expressão com valor verdade* é uma expressão que retorna um valor verdade equivalente a (`True` ou `False`)¹.

Por exemplo

```
if idade>18:  
    print("Pode_dirigir")
```

Muitas decisões tem dois caminhos: devemos fazer algo se a expressão for verdadeira e outra coisa se a expressão for falsa. Isso é conhecido como *se-então*.

Por exemplo

```
if idade>18:  
    print("Pode_dirigir")  
else:  
    print("Proibido_dirigir")
```

Algumas vezes temos várias opções, então temos vários testes que devem ser feitos. Isso é possível com a expressão `if-elif-else`

Por exemplo

```
if idade>18:  
    print("Pode_dirigir")  
elif idade>90:  
    print("Deve_evitar_dirigir")  
else:  
    print("Proibido_dirigir")
```

Também é possível fazer um `if` dentro do outro.

Por exemplo:

```
if idade>18:
```

¹ Qualquer valor que possa ser interpretado como um valor verdade ??

```

print("Pode_dirigir")
if sexo=="masculino":
    print("Deve_possuir_certificado_de_reservista")
else:
    print("Liberada_de_ser_reservista")
else:
    print("Proibido_dirigir")

```

11.1 Expressões Condicionais

Python possui um operador baseada no conceito de `if` que pode ser usado em expressões. A sintaxe é:

<valor se verdade> **if** <expressão> **else** <valor se falso>

Por exemplo, o programa a seguir:

```

a = 1 if True else 2
b = 1 if True else 2
c = 7 if 10>8 else 12
print(a,b,c)

```

Resulta em:

```
1 2 7
```

11.2 Problemas com If

Exercício 18. Um cinema de Patópolis tem que adaptar seu software à nova lei de meia-entrada do seu país. Basicamente, a meia entrada tem em Patópolis a seguinte regra:

- Idosos, maiores de 60 anos, sempre pagam meia entrada.
- Crianças, menores de 12 anos, sempre pagam meia entrada.
- Adolescentes que são estudantes, entre 12 (inclusive) e 17 (inclusive) anos, SEMPRE pagam meia entrada.
- Outros estudantes pagam meia entrada enquanto não forem vendidos 40% dos ingressos. Caso contrário, pagam inteira.
- Todos os outros casos pagam a entrada inteira.

Escreva uma função em Python `valor_entrada` que retorne o valor a ser pago quando alguém for comprar uma entrada. Seus parâmetros devem ser 5:

- `valor_integral` - valor integral da entrada, em float
- `idade` - idade do comprador, em int
- `estud` - True se for estudante, False se não for
- `bil_vendidos` - quantidade de bilhetes vendidos até o momento
- `bil_avenda` - quantidade total de bilhetes à venda


```
def valor_entrada(val_int, idade, estud, bil_vendidos, bil_avenda):  
    if idade<12 or idade >= 60:  
        return val_int*0.5  
    elif 12<idade<18 and estud:  
        return val_int*0.5  
    elif estud and (1.0*bil_vendidos/bil_avenda < 0.4):  
        return val_int*0.5  
    else:  
        return val_int
```


12

Funções Recursivas

Funções podem chamar a si mesmas. Porém, para que isso possa acontecer com algum sentido, é necessário que elas, em algum momento, parem de fazer isso e retornem algum resultado. Isso é feito por meio de uma decisão, ou seja, um `if` no código.

Com funções recursivas podemos resolver facilmente uma infinidade de problemas.

O exemplo mais famoso é o da função fatorial. O fatorial de um número inteiro é o produto de todos os números inteiros positivos até ele. O fatorial de 1 é 1, o fatorial de 2 é 2, o fatorial de 3 é 6, o fatorial de 4 é 24. Por definição o fatorial de 0 é 1. Uma maneira recursiva de definir fatorial é dizer que o fatorial de um número é o produto dele vezes o fatorial do número anterior, até que esse número seja 1 ou 0. O programa a seguir calcula o fatorial de qualquer número natural.

```
def fatorial(n):  
    if n==0 or n==1:  
        return 1  
    else:  
        return n*fatorial(n-1)
```

Outro exemplo muito conhecido é a Sequência de Fibonacci, definida da seguinte forma: $F(1) = 1, F(2) = 1$ e $F(n) = F(n-1) + F(n-2)$. Ela pode ser programada facilmente como:

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

12.1 Recursividade e String

A notação de índice e fatia permite que resolvamos problemas interessantes com string¹. Isso porque sempre podemos separar uma string em duas partes, sua primeira letra e as outras letras. Assim podemos processar a string letra a letra, sem nenhum problema maior.

¹ E mais tarde com outras sequências

Por exemplo, suponho que você deseja fazer um texto cifrado, trocando cada letra por sua letra seguinte no alfabeto. Primeiro podemos fazer uma função que faz isso para uma letra:

```

1 def troca(letra):
2     if letra == "Z":
3         return "A"
4     elif letra == "z":
5         return "a"
6     elif ord("a")<=ord(letra)<ord("z") :
7         return chr(ord(letra)+1)
8     elif ord("A")<=ord(letra)<ord("Z") :
9         return chr(ord(letra)+1)
10    else:
11        return letra

```

Nesse programa tomamos alguns cuidados, até excessivos, como usar o `elif`, já que damos o retorno imediatamente, e separar as linhas 6 e 8 para melhor clareza, quando o mais correto seria usar:

```

elif ord("a")<=ord(letra)<ord("z") or ord("A")<=ord(letra)<ord("Z") :

```

Agora vamos usar essa função na nossa cifra:

```

12 def cifra(s):
13     if s=="":
14         return s
15     else:
16         return troca(s[0])+cifra(s[1:])

```

O que estamos fazendo é trocar a primeira letra da string e depois concatenar com a cifra do resto da string. Isso vai acontecer até que o resto da string seja vazio, o final obrigatório da recursão.

A linha 2 também podia fazer a pergunta `if len(s)==0:`.

O programa acima pode ser modificado para imprimir algumas coisas e indicar o que está acontecendo.

```

def cifra2(s,nivel=0):
    print(""*nivel,nivel,":chamou_s=_",s)
    if s=="":
        print(""*nivel,nivel,":retorna_s=_",s)
        return s
    else:
        r1,r2 = troca(s[0]),cifra2(s[1:],nivel+1)
        print(""*nivel,nivel,":vai_retornar_r1=_",r1,"+", "r2=_",r2)
        return r1+r2
print(cifra2("senha"))

```

E o resultado é, para a execução `cifra2("senha")`:

```

0 :chamou s= senha
* 1 :chamou s= enha
** 2 :chamou s= nha
*** 3 :chamou s= ha
**** 4 :chamou s= a
***** 5 :chamou s=
***** 5 :retorna s=
**** 4 :vai retornar r1= b + r2 =
*** 3 :vai retornar r1= i + r2 = b
** 2 :vai retornar r1= o + r2 = ib
* 1 :vai retornar r1= f + r2 = oib
0 :vai retornar r1= t + r2 = foib
tfoib

```

12.2 Problemas com Recursividade

Exercício 19. Faça uma função `elimina(s,c)` que elimine todos os caracteres `c` de uma string `s`, sendo que por default deve eliminar espaços.

```

1 def elimina(s,c="_"):
2     if s=="":
3         return s
4     elif s[0]==c:
5         return elimina(s[1:],c)
6     else:
7         return s[0]+elimina(s[1:],c)

```

Veja que temos que lembrar de passar o argumento `c` nas chamadas recursivas, ou na próxima chamada `c` assumirá o valor *default* de espaço (linhas 5 e 7). Além disso, atenção como na linha 5 simplesmente continuamos a função sem colocar nada no lugar de `s[0]`, eliminando esse carater, e na linha 7, mesmo não fazendo alteração nenhuma no `s[0]` temos que garantir que ele será concatenado com uma string onde a eliminação foi feita.

13

Repetindo instruções com while

É comum que tenhamos que fazer o mesmo processamento, ou processamentos similares, várias vezes. Por exemplo, calcular a média de vários alunos, ou buscar um resultado cada vez com um erro menos, ou tentar várias soluções possíveis até que uma funcione.

Vamos imaginar um exemplo. Suponha que você quer ensinar a um motorista a chegar a um lugar cujo caminho é:

- vire a direita
- vire a direita
- vire a direita

Seria simples dizer isso, porém pode ser mais simples ainda dizer “vire a direita três vezes”.

Da mesma forma como podemos decidir se vamos fazer ou não um bloco, podemos também repetir um bloco várias vezes, decidindo, a cada passagem pelo bloco, se vamos parar ou não.

Para isso usamos o comando `while`, cuja tradução é **enquanto**. Sua forma mais básica é:

```
while <expressão lógica>:  
    <bloco>
```

Veja que usamos de novo a indentação para dizer que o bloco está subordinado ao comando.

Uma forma simples de usar o `while` é quando temos que repetir algo um número fixo de vezes. Por exemplo, se desejarmos calcular o fatorial de um número, o seguinte programa pode ser usado:

```
1 numero = 21  
2 fatorial = 1  
3 i = 1  
4 while i <= numero:  
5     fatorial = fatorial * i  
6     i = i+1  
7 print(fatorial)
```

Resulta em

51090942171709440000

13.1 A condição

O `while` continua, ou seja, o seu bloco continua sendo executado sempre que a expressão estipulada for verdade. Isso significa que precisamos garantir algumas coisas para que ele funcione bem:

- A expressão deve ter um valor válido na sua primeira execução, verdadeiro ou falso¹.
- Eventualmente, a expressão deve mudar para um valor falso.

No programa anterior, a expressão é `i <= numero`. Para que ela seja calculável é importante que `i` e `numero` estejam definidos. Isso pode ser visto nas linhas 1 e 3.

A condição se altera na linha 6. Quando somamos 1 a `i` fazemos com que ele aumente. Se `numero` nunca for alterado no bloco, e isso não acontece mesmo, então eventualmente `i` será maior que `numero` e o termina.

¹ Normalmente o valor é verdadeiro pelo menos uma vez, mas existem programas onde o `while` nunca é executado. Nesse exemplo, isso aconteceria se `numero` tivesse o valor de 1

13.2 Contadores

O programa anterior é um exemplo do uso de contadores. Um contador é uma variável inteira a qual somamos normalmente 1.

13.3 Programas que nunca param

É comum errarmos a condição, ou esquecermos de mudar a condição. O erro mais comum é esquecermos de somar 1 ao contador.

13.4 `break`

Algumas vezes queremos, por algum motivo, parar a execução do bloco em algum ponto intermediário. Para isso existe a instrução `break`. Ao encontrar um `break` o `while` mais interno imediatamente acaba.

É comum usar o `break` junto com a condição `while True`:. Veja-mos um exemplo:

```
1 # esse programa funciona até encontrar um número divisível por 6
2 from random import randint
3 while True:
4     numero = randint(1,100)
5     if numero % 6 == 0:
6         break
7 print(numero)
```

Esse programa é equivalente a:

```
1 # esse programa funciona até encontrar um número divisível por 6
2 from random import randint
3 numero = randint(1,100)
4 while numero % 6 != 0:
```



```

5     numero = randint(1,100)
6     print(numero)

```

Qual a equivalência. Os primeiro programa não precisa fazer a primeira conta feita no segundo programa (linha 3), porque ela só existe para garantir que o `while` tem uma expressão para avaliar. Além disso, a linha 4 do primeiro programa **sempre** será executada, enquanto a linha 5 do segundo programa pode ser ou não ser executada, dependendo do resultado da expressão anterior.

A construção:

```

1  while True:
2      <bloco>
3      if <expressão com valor verdade>:
4          break

```

É, na prática, o raciocínio conhecido como *repita-até*, ou em inglês, *repeat until*².

O Programa:

```

1  j = 0
2  i = 0
3  while True:
4      i = i + 1
5      j = j + i
6      if i == 100:
7          break

```

Soma todos os números de 1 a 100.

² Várias linguagens de programação implementam diretamente uma versão do comando `repeat <bloco> until <expressão>`. Essa instrução não existe em Python.

13.5 *continue*

`Continue` é executado para continuar no loop, porém não executar os passos a seguir, voltando ao início do loop.

```

1  while True:
2      <bloco>
3      if <expressão com valor verdade>:
4          continue
5      <mais bloco>

```

Por exemplo, imprimir os números de 1 a 100, menos os múltiplos de 6.

```

1  i = 0
2  while i <= 100:
3      i = i + 1
4      if i % 6 == 0:
5          continue
6      print(i)

```

13.6 *while também tem else*

A instrução `else` também pode ser usada com `while`. Após executar seu último ciclo, ou seja, quando a condição do `while` é falsa, o bloco subordinado ao `else` é executado.

```

1 i = 1
2 while i <= 100:
3     if i % 6 == 0:
4         continue
5     print(i)
6     i = i + 1
7 else:
8     print("Acabou")

```

O `else` do `while` não é ativado quando ocorre um `break`.

O programa abaixo, por exemplo, não imprime a frase `fez else`

```

i = 0
print("Inicia")
while i <= 3:
    i = i + 1
    print("i_=_", i)
    if i >= 2:
        print("Parou_no_meio")
        break
else:
    print("Fez_else")
print("Acaba")

```

Resultando em:

```

Inicia
i = 1
i = 2
Parou no meio
Acaba

```

É interessante notar que `else` são pouquíssimo usados em `whiles`.

13.7 *Problemas com While*

Exercício 20. Faça uma função que retorne o valor de $\sum_{i=1}^n i$ para um valor de `n` passado como parâmetro.

```

def somatorio(n):
    i = 0
    soma = 0
    while i <= n:
        soma = soma + i
        i = i + 1
    return soma

```

Esse problema é dos mais típicos. Ele mostra a necessidade de inicializar duas variáveis: `i` é usada para passar por todos os números que precisamos, enquanto `soma` é usada para guardar o resultado da soma, somando a si mesma cada passo.

14

Tuplas

Uma tupla é uma sequência ordenada de objetos, que não precisam ser do mesmo tipo. Elas são indicadas pelos separadores ().

Um tupla de números inteiros, por exemplo, pode ser (1,3,1). Como vemos, não há nenhuma limitação de quantidade de vezes que o número aparece, Enquanto a tupla existe e não é modificada, a ordem dos tupla da tupla é mantida. Uma tupla pode ser de tipos variados, como em ["string", "A", 12, 213.0+129JJ].

Os principais usos para tuplas são representar:

- vetores
- posições em 2D ou 3D
- pequeno conjunto de propriedades de algo ou alguém

Uma tupla vazia é criada de forma muito simples:

```
tupla = (,)
```

Usamos essa vírgula na tupla vazia para separá-la da expressão abrir e fechar parênteses.

Uma tupla já com alguns membros pode ser criada diretamente:

```
tupla = (1,2.0,"nome")
```

Também podemos criar tuplas e guardar em variáveis sem usar os parênteses. Python consegue entender isso.

O programa:

```
tupla = 1,2,3  
print(tupla)
```

Dá o resultado:

```
(1, 2, 3)
```

Também podemos criar tuplas a partir de outros tipos, usando a função tuple(x). Por exemplo, uma tupla de letras pode ser criada a partir de uma string.

O programa:

```
letras = tuple("Uma_string_vira_tupla")  
print(letras)
```

Dá o resultado:

```
('U', 'm', 'a', ' ', 's', 't', 'r', 'i', 'n', 'g', ' ', 'v', 'i', 'r', 'a', ' ', 't', 'u', 'p', 'l', 'a')
```

Podemos perguntar o tamanho de uma tupla com a função `len(x)` que usamos em strings. A finalidade dela é calcular o tamanho da tupla. Atenção porque o último elemento da tupla é o tamanho da tupla-1, já que a tupla começa em zero.

O programa:

```
tupla = (1, "alfa", 3.0 )
print(len(tupla))
```

Dá o resultado:

```
3
```

14.1 Índices e Fatias em Tuplas

Da mesma forma como trabalhos com strings, podemos indexar itens específicos de uma tupla ou ainda fatias.

O programa:

```
tupla1 = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
print("tupla1[0]_=_", tupla1[0])
print("tupla1[5]_=_", tupla1[5])
print("tupla1[-1]_=_", tupla1[-1])
print("tupla1[5]_=_", tupla1[-5])
print("tupla1[-10]_=_", tupla1[-10])
print("tupla1[:]_=_", tupla1[:])
print("tupla1[3:]_=_", tupla1[3:])
print("tupla1[:5]_=_", tupla1[:5])
print("tupla1[2:6]_=_", tupla1[2:6])
print("tupla1[::2]_=_", tupla1[::2])
print("tupla1[2:9:3]_=_", tupla1[2:9:3])
print("tupla1[5:1:-1]_=_", tupla1[5:1:-1])
print("tupla1[9:2:-2]_=_", tupla1[9:2:-2])
print("tupla1[::-1]_=_", tupla1[::-1])
```

Dá o resultado:

```
tupla1 = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
tupla1[0] = 1
tupla1[5] = 6
tupla1[-1] = 10
tupla1[5] = 6
tupla1[-10] = 1
tupla1[:] = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
tupla1[3:] = (4, 5, 6, 7, 8, 9, 10)
tupla1[:5] = (1, 2, 3, 4, 5)
tupla1[2:6] = (3, 4, 5, 6)
tupla1[::2] = (1, 3, 5, 7, 9)
```

```
tupla1[2:9:3] = (3, 6, 9)
tupla1[5:1:-1] = (6, 5, 4, 3)
tupla1[9:2:-2] = (10, 8, 6, 4)
tupla1[::-1] = (10, 9, 8, 7, 6, 5, 4, 3, 2, 1)
```

Outros operadores de uso comum de tuplas é o `in` e o `not in`. Eles verificam se um item está em uma tupla.

O programa:

```
cores = ("vermelho", "azul", "verde", "amarelo")
print("_As_cores_existentes:_", cores)
pedido = input("Escolha_sua_cor:_")
existe = pedido in cores
print("Seu_pedido_existe?_", existe)
```

Dá o resultado:

```
As cores existentes: ('vermelho', 'azul', 'verde', 'amarelo')
```

```
Escolha sua cor: vermelho
```

```
Seu pedido existe? True
```

Também é possível concatenar tuplas, como fazemos com strings.

O programa:

```
tupla1 = (1, 2, 3, 4, 5)
tupla2 = (6, 7, 8, 9, 10)
print(tupla1+tupla2)
```

Dá o resultado:

```
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

E, mais uma vez, se precisamos inicializar uma tupla para ela ter o tamanho que precisamos, podemos usar o operador `*` para fazer isso.

O programa:

```
tupla1 = (0)*5
tupla2 = (6, 7)*3
print(tupla1,tupla2)
```

Dá o resultado:

```
0 (6, 7, 6, 7, 6, 7)
```

14.2 tuplas de tuplas

É possível colocar qualquer objeto dentro de tuplas, inclusive tuplas.

O programa:

```
tupla1 = (0, (2,4), 5)
print(tupla1)
```

Dá o resultado:

```
(0, (2, 4), 5)
```

14.3 O operador *in*

Tuplas, como todas as sequências, possui um operador *in* que diz se um elemento está dentro dela. Também é possível verificar se um elemento não pertence a uma tupla usando o operador *not in*.

```
>>> 1 in (1,2,3,4)
True
>>> 5 in (1,2,3,4)
False
>>> 1 not in (1,2,3,4)
False
>>> 5 not in (1,2,3,4)
True
```


Listas - *list*

Como o nome diz, o tipo de dado `list` fornece listas. Uma lista é um sequência ordenada de objetos, que não precisam ser do mesmo tipo. Elas são indicadas pelos separadores `[]`, o mesmo que usamos para construir índices e fatias, mas podem ser facilmente distinguidas pelo uso.

Grande parte do comportamento de listas é igual ao comportamento de tuplas. Isso acontece porque ambas são sequências em Python¹. A grande diferença é que listas são mutáveis. Isso significa que você pode alterar o interior de uma lista sem trocar a tupla propriamente dita.

Um lista de números inteiros, por exemplo, pode ser `[1, 3, 5, 9, 12, 4, 2, 4, 1, 1, 1]`. Como vemos, não há nenhuma limitação de quantidade de vezes que o número aparece, Enquanto a lista existe e não é modificada, a ordem dos itens da lista é mantida. Uma lista pode ser de tipos variados, como em `["string", "A", 12, 213.0+129JJ]`.

Os principais usos para listas são representar:

- vetores
- matrizes pequenas
- itens que pertencem a algo ou alguém
- coleções com repetições
- itens que devem ser, ou já foram, processados
- pequeno conjunto de propriedades de algo ou alguém

Uma lista vazia é criada de forma muito simples:

```
lista = []
```

Uma lista já com alguns membros pode ser criada diretamente:

```
lista = [1, 2.0, "nome"]
```

Também podemos criar listas a partir de outros tipos, usando a função `list(x)`. Por exemplo, uma lista de letras pode ser criada a partir de uma string.

O programa:

Listas são uma das forças de Python como linguagem. Elas formam uma estrutura básica que é incomum de ser integrada a linguagens imperativas, mas que são encontradas em linguagens funcionais e lógicas.

¹ Junto com `range`

```
lista = list("Uma_string_vira_lista")
print(lista)
```

Dá o resultado:

```
['U', 'm', 'a', ' ', 's', 't', 'r', 'i', 'n', 'g', ' ', 'v', 'i', 'r', 'a', ' ', 'l', 'i', 's', 't', 'a']
```

A função mais usada em listas é provavelmente a mesma função `len(x)` que usamos em strings. A finalidade dela é calcular o tamanho da lista. Atenção porque o último elemento da lista é o tamanho da lista-1, já que a primeira posição da lista começa em zero.

O programa:

```
lista = [1, "alfa", 3.0 ]
print(len(lista))
```

Dá o resultado:

3

15.1 Índices e Fatias em Listas

Da mesma forma como trabalhamos com strings, podemos indexar itens específicos de uma lista ou ainda fatias.

A lista `[1,5.0,"alfa",4,45,"d",7,9,7,9.0]` tem as posições como indicadas na Figura 15.1.

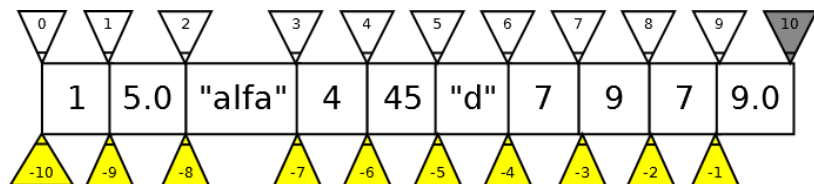


Figura 15.1: Posição dos índices em listas

O programa:

```
lista1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print("lista1[0]=", lista1[0])
print("lista1[5]=", lista1[5])
print("lista1[-1]=", lista1[-1])
print("lista1[5]=", lista1[-5])
print("lista1[-10]=", lista1[-10])
print("lista1[:]=", lista1[:])
print("lista1[3:]=", lista1[3:])
print("lista1[:5]=", lista1[:5])
print("lista1[2:6]=", lista1[2:6])
print("lista1[::2]=", lista1[::2])
print("lista1[2:9:3]=", lista1[2:9:3])
print("lista1[5:1:-1]=", lista1[5:1:-1])
print("lista1[9:2:-2]=", lista1[9:2:-2])
print("lista1[::-1]=", lista1[::-1])
```

Dá o resultado:

```
lista1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
lista1[0] = 1
lista1[5] = 6
lista1[-1] = 10
lista1[5] = 6
lista1[-10] = 1
lista1[:] = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
lista1[3:] = [4, 5, 6, 7, 8, 9, 10]
lista1[:5] = [1, 2, 3, 4, 5]
lista1[2:6] = [3, 4, 5, 6]
lista1[::2] = [1, 3, 5, 7, 9]
lista1[2:9:3] = [3, 6, 9]
lista1[5:1:-1] = [6, 5, 4, 3]
lista1[9:2:-2] = [10, 8, 6, 4]
lista1[::-1] = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

15.2 Alterando a Lista

Os operadores de índice e de fatia também podem ser usados para alterar o valor da lista²

Para isso, basta dizer que pedaço da lista você quer mudar.

O programa:

```
lista1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(lista1)
lista1[0]=4
print(lista1)
lista1[1]=[5,5]
print(lista1)
lista1[-1]=433
print(lista1)
lista1[2:7]=[5,6,7]
print(lista1)
lista1[:2]=[51,61,71]
print(lista1)
lista1[5:]=[4,8]
print(lista1)
lista1[2:7:2]=[5,6,7]
print(lista1)
lista1[8:1:-3]=[5,6]
print(lista1)
```

Dá o resultado:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[4, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[4, [5, 5], 3, 4, 5, 6, 7, 8, 9, 10]
[4, [5, 5], 3, 4, 5, 6, 7, 8, 9, 433]
[4, [5, 5], 5, 6, 7, 8, 9, 433]
```

² Lembramos que isso não é possível com tuplas.

```
[51, 61, 71, 5, 6, 7, 8, 9, 433]
[51, 61, 71, 5, 6, 4, 8]
[51, 61, 5, 5, 6, 4, 7]
[51, 61, 5, 6, 6, 4, 5]
```

15.3 Operadores

Outros operadores de uso comum de listas é o `in` e o `not in`. Eles verificam se um item está em uma lista.

O programa:

```
cores = ["vermelho", "azul", "verde", "amarelo"]
print("_As_cores_existentes:_", cores)
pedido = input("Escolha_sua_cor:_")
existe = pedido in cores
print("Seu_pedido_existe?_", existe)
```

Dá o resultado:

```
As cores existentes: ['vermelho', 'azul', 'verde', 'amarelo']
```

```
Escolha sua cor: vermelho
```

```
Seu pedido existe? True
```

Também é possível concatenar listas, como fazemos com strings.

O programa:

```
lista1 = [1, 2, 3, 4, 5]
lista2 = [6, 7, 8, 9, 10]
print(lista1+lista2)
```

Dá o resultado:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

E, mais uma vez, se precisamos inicializar uma lista para ela ter o tamanho que precisamos, podemos usar o operador `*` para fazer isso.

O programa:

```
lista1 = [0]*5
lista2 = [6, 7]*3
print(lista1, lista2)
```

Dá o resultado:

```
[0, 0, 0, 0, 0] [6, 7, 6, 7, 6, 7]
```

15.4 Listas de Listas

É possível colocar qualquer objeto dentro de listas, inclusive listas.

O programa:

```
lista1 = [0,[2,4],5]
print(lista1)
```

Dá o resultado:

```
[0, [2, 4], 5]
```

15.5 Representando Matrizes

Matrizes podem ser representadas como listas de listas. Por exemplo, a matriz:

$$M = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 & 7 \end{bmatrix}$$

Pode ser escrita em Python como:

```
M = [[1, 2, 3, 4, 5], [3, 4, 5, 6, 7]]
```

O raciocínio é que cada item da lista é uma linha da matriz. Todas as listas devem ter o mesmo tamanho, e cada posição da lista é uma célula da matriz.

A notação para acessar uma célula é `M[i][j]`, onde `i` é a linha e `j` a coluna.

Mais tarde vamos ver como fazer operações com as matrizes usando os comandos `while` e `for`.

15.6 O operador `in`

Listas, como todas as sequências, possui um operador `in` que diz se um elemento está dentro dela. Também é possível verificar se um elemento não pertence a uma lista usando o operador `not in`.

```
>>> 1 in [1,2,3,4]
True
>>> 5 in [1,2,3,4]
False
>>> 1 not in [1,2,3,4]
False
>>> 5 not in [1,2,3,4]
True
```

15.7 Apagando Elementos de Uma Lista

Para retirar um elemento de uma lista se usa o operador `del` e se indica que elemento tirar usando uma notação de índice ou de fatia.

```
>>> l=[1,2,3,4,5,6]
>>> del l[2]
>>> l
[1, 2, 4, 5, 6]
```

```
>>> del l[:2]
>>> l
[2, 5]
```

15.8 Métodos de Lista

Vários métodos podem ser aplicados a listas. A lista completa de métodos pode ser encontrada em

- `list.append(l,i)`, anexa (*append*) um item *i* ao fim da lista *l*
- `list.extend(l1,l2)`, estende a lista *l1* com os elementos da lista *l2*
- `list.insert(l,p,e)`, insere elemento *e* na posição *p* da lista *l*. Lembre que posições são melhor entendidas como o intervalo entre os itens.
- `list.remove(l,e)`, remove a primeira aparição de *e* em *l*, causa um erro se não existir
- `list.pop(l,p)` remove e retorna o elemento na posição *p* sem causar erros. Se *p* não existir, assume o último elemento.
- `list.count(l1,l2,inicio,fim)`, retorna quantas vezes *l2* aparece em *l1*, entre as posições *inicio* e *fim*-1.
- `list.index(l1,l2,inicio,fim)`, retorna a posição da primeira aparição de *l2* em *l1* entre *inicio* e *fim*-1. Causa um erro se *l2* não está em *l1*
- `reverse(l)`, reverte a lista
- `sort(l)`, ordena a lista.

15.9 Problemas com Listas

Exercício 21. Faça uma função em Python, `divisores(n)`, que retorne a lista de divisores de um número natural *n*.

```
def divisores(n):
    i = 1
    divs = []
    while i <= n:
        if n % i == 0:
            divs.append(i)
        i = i + 1
    return divs
```

Dicionários

Dicionários permitem que seja mantida uma estrutura de dados que associa chaves a valores.

Dois exemplos típicos de dicionários na vida real: os dicionários propriamente ditos tem como chaves as palavras, e como valores suas explicações, as agendas telefônicas tem como chave o nome da pessoa e como valor os seus telefones.

Dicionários são indicados pelo abrir e fechar chaves, tendo seus pares chave-valor separados por dois pontos ¹. Seguem exemplos:

```
1 {}
2 { "novo" : 5 , "velho" : 10}
3 { "Ana" : "555-4232", "Beto" : "123-1223", 12 : 12.0 }
4 dict()
```

¹ Conjuntos, do tipo set, também usam a notação de abrir e fechar chaves, porém os itens são simples e não compostos como um par chave-valor. A única maneira de escrever um conjunto vazio é usar o construtor set

A primeira linha mostra um dicionário vazio. A segunda linha mostra um dicionário onde todas as chaves são strings e todos os valores são inteiros. A terceira linha mostra que dicionários podem ter chaves e valores de tipos diferentes e finalmente a quarta linha mostra a função construtora de dicionários, que retorna um dicionário vazio.

As chaves de um dicionário tem que ser de um tipo imutável. O mais comum é que usemos os dicionários quando nossas chaves são strings, mas podem ser tuplas ou outro tipo imutável. Não podem, porém, ser listas ou outros dicionários, que são tipos mutáveis.

O valor pode ser de qualquer tipo. É comum que sejam listas, como em uma agenda telefônica que permite vários números.

Dicionários não mantêm a ordem de seus itens.

Colocamos valores em um dicionário usando o operador []. Assim, podemos ir construindo um dicionário passo a passo, iniciando com um dicionário vazio, como em:

```
d = {}
d["novo"] = 5
d["velho"] = 10
```

Ao fazer isso, chave e o valor são inseridos no dicionário. Se a chave já existe, o valor é sobreescrito. Se a chave não existe, seu valor é criado.

Usamos a mesma notação para recuperar valores de um dicionário. O programa a seguir:

```
d = {}
d["novo"] = 5
d["velho"] = 10
print(d["novo"])
```

Resulta em:

```
5
```

Ao acessar um valor de um dicionário com a notação de colchetes sempre temos que fornecer um valor válido, uma chave existente no dicionário, ou receberemos um erro, como em:

```
>>> d={}
>>> d[1]=5
>>> d
{1: 5}
>>> d[2]
```

Traceback (most recent call last):

```
File "<pyshell#13>", line 1, in <module>
    d[2]
```

KeyError: 2

A função `dict` cria dicionários. Se nenhum argumento for passado, ela cria um dicionário vazio. Existem porém duas outras formas de criar dicionários já com alguns elementos: passar uma lista de pares chave valor ou cada argumento na forma de uma atribuição.

Exemplo da primeira forma, como lista de pares, podemos usar indiferentemente tuplas ou listas:

```
idades = dict([["Ana", 14], ["Beto", 18], ["Carlos", 18]])
olhos = dict([["Ana", "azuis"], ("Beto", "castanhos"), ("Carlos", "negros")])
print(idades)
print(olhos)
```

Resulta em:

```
'Beto': 18, 'Ana': 14, 'Carlos': 18
'Beto': 'castanhos', 'Ana': 'azuis', 'Carlos': 'negros'
```

Com atribuições, o uso tem a forma:

```
cartas = dict(Ana=["3P", "3E", "40", "QE", "QC"], Beto="sem_cartas")
print(cartas)
```

Resulta em:

```
'Beto': 'sem cartas', 'Ana': ['3P', '3E', '40', 'QE',
'QC']
```

16.1 Funções de dicionário

As operações mais usadas com dicionário são:

- `len(d)`, retorna o tamanho do dicionário (quantas chaves)
- `dict.keys(d)`, retorna uma lista com todas as chaves
- `dict.values(d)`, retorna uma lista com todos os valores
- `dict.items(d)`, retorna uma lista com todos os pares chave-conteúdo do dicionário
- `dic.get(d, c, r)`, retorna o valor da chave `c` no dicionário `d`, se a chave não está presente retorna `r`. Se `r` não for especificado, retorna `False`
- `dict.clear(d)`, limpa o dicionário, deixando ele vazio
- `dict.copy(d)`, retorna uma cópia do dicionário

Todas essas funções podem ser usadas diretamente como métodos, isto é, usando a notação de ponto.

```
idades = dict([["Ana", 14], ["Beto", 18], ["Carlos", 18]])
idades2 = idades.copy()
print(idades)
print(idades2)
idades2["Julia"] = 9
print(idades)
print(idades2)
idades["Ana"] = 15
print(idades)
print(idades2)
```

Resulta em:

```
'Beto': 18, 'Ana': 14, 'Carlos': 18
'Beto': 18, 'Ana': 14, 'Carlos': 18
'Beto': 18, 'Ana': 14, 'Carlos': 18
'Julia': 9, 'Beto': 18, 'Ana': 14, 'Carlos': 18
'Beto': 18, 'Ana': 15, 'Carlos': 18
'Julia': 9, 'Beto': 18, 'Ana': 14, 'Carlos': 18
```


17

Conjuntos

Python possui um tipo `set` que determina um conjunto. A principal característica de um conjunto é que ele é uma lista **sem** valores repetidos.

Um conjunto é representado por itens separados por vírgula, dentro de parênteses, como no conjunto de vogais: `{"a", "e", "i", "o", "u"}`¹

Ao contrário de outros tipos, não existe uma representação visual para o conjunto vazio².

Conjuntos são coleções não ordenadas. Isso significa que não há garantia que a ordem se mantém entre uma utilização e outra de um conjunto.

¹ `set` e `dict` compartilham o mesmo delimitador, mas dicionários precisam ter elementos com chaves e valores separados por dois pontos.

² Ela se confundiria com a representação visual de um dicionário vazio, previamente existente

17.1 O operador `in`

Conjuntos, como todas as sequências, possui um operador `in` que diz se um elemento está dentro dele. Também é possível verificar se um elemento não pertence a um conjunto usando o operador `not in`.

```
>>> 1 in [1,2,3,4]
True
>>> 5 in [1,2,3,4]
False
>>> 1 not in [1,2,3,4]
False
>>> 5 not in [1,2,3,4]
True
```

17.2 Operadores com conjuntos

Python fornece vários operadores para conjuntos, descritos na Tabela 17.1.

Operador	Significado
<code>e in A</code>	pertinência
<code>e not in A</code>	negação da pertinência
<code>A B</code>	união dos conjuntos
<code>A & B</code>	interseção dos conjuntos
<code>A - B</code>	Elementos que pertencem a A e não a B
<code>A ^ B</code>	$(A B) - (A \& B)$
<code>A < B</code>	A contido em B
<code>A <= B</code>	A contido e diferente de B
<code>A <= B</code>	A contido ou igual a B
<code>A > B</code>	A contém e é diferente de B
<code>A >= B</code>	A contém ou é igual a B

Tabela 17.1: Operadores de conjuntos para elemento e e conjuntos A e B

```
>>> s1 = {1,2,3,4}
>>> s2 = {3,4,5,6}
>>> s1-s2
set([1, 2])
>>> s2-s1
set([5, 6])
>>> s1 | s2 # união
set([1, 2, 3, 4, 5, 6])
>>> s1 ^ s2 # diferença simétrica
set([1, 2, 5, 6])
>>> s1 & s2 # interseção
set([3, 4])
>>> s1 < s2 # está contido
False
>>> s3 = {2, 3}
>>> s3 < s1
True
>>> s3 < s2
False
```

17.3 Funções que alteram um set

- `set.add(s,e)`, adiciona o elemento e no conjunto s
- `set.remove(s,e)`, remove o elemento e do conjunto s, acontece um erro se e não está em s
- `set.discard(s,e)`, remove o elemento e do conjunto s sem gerar erros.
- `set.clear(s)`, limpa o conjunto s, tornando-o vazio
- `set.pop(s)`, retorna um elemento qualquer do conjunto, tira o elemento do conjunto, causa um erro ao encontrar um conjunto vazio.

17.4 *frozenset*

Enquanto um `set` é mutável, um `frozenset` é imutável.

Mutabilidade

Nos capítulos de tuplas e listas vimos duas estruturas de sequência muito paradas. A principal diferença entre elas, além da notação, é que tuplas são imutáveis e listas são mutáveis.

O que isso realmente significa.

Primeiro precisamos entender como funcionam as variáveis em Python. Em Python, as variáveis são nomes que indicam um valor na memória. Esses valores são objetos de uma classe (ou tipo) de Python. Alguns valores são simples, como os do tipo `int`, outros são mais complexos, como os do tipo `list`.

Muitas vezes chamamos estes tipos por tipos simples e tipos estruturados.

Bem, quando um tipo permite que o valor seja alterado internamente sem alterar a identidade do valor, dizemos que ele é mutável. Quando isso é impossível, dizemos que ele é imutável.

Isso fica bem claro com o operador de índice e fatias, `[]`. Strings e tuplas são imutáveis, isso quer dizer que só podemos calcular letras ou itens que estão em uma posição ou pertencem a uma fatia. Por outro lado, listas são mutáveis. Isso significa que podemos alterar um item ou um pedaço da lista que está em uma posição ou fatia.

A melhor ferramenta para entender isso é a Python Tutor ©, acessível em <http://www.pythontutor.com/>.

Objetos simples e imutáveis são representados diretamente em frames. Objetos estruturados ou mutáveis já precisam ser estruturados na memória de objetos.

```
Python 3.3
→ 1 a = [1, 2, 3]
   2 b = [0] * 7
   3 b[2] = a
   4 a[1] = 3
   5 print(b)
```

Figura 18.1: Primeiro passo, definir a como uma lista

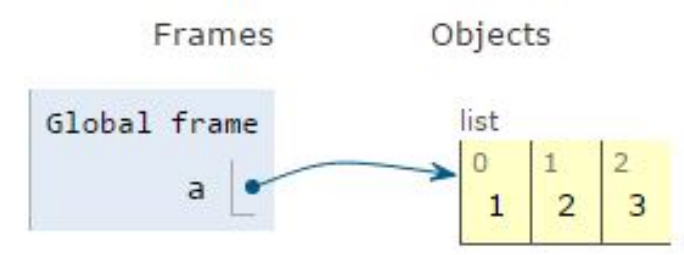


Figura 18.2: Na memória podemos ver que o nome aponta para uma lista

```
Python 3.3
→ 1 a =[1,2,3]
→ 2 b = [0]*7
  3 b[2] = a
  4 a[1] = 3
  5 print(b)
```

Figura 18.3: Segundo passo, definir b como uma lista de zeros

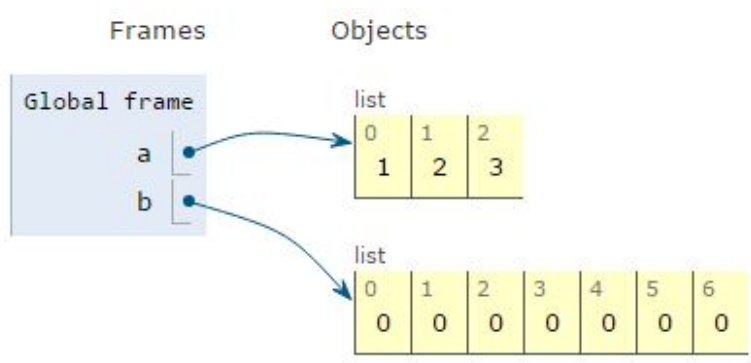


Figura 18.4: Uma nova lista é criada para b

```
Python 3.3
  1 a =[1,2,3]
→ 2 b = [0]*7
→ 3 b[2] = a
  4 a[1] = 3
  5 print(b)
```

Figura 18.5: Terceiro passo, definir colocar a dentro de b

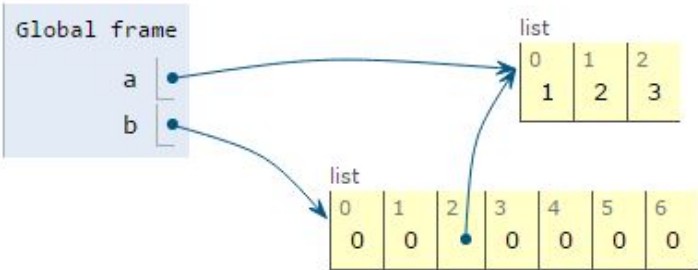


Figura 18.6: A lista a agora está dentro da lista b. Veja que ela não é copiada para dentro, mas apontada por uma referência. O programador não pode acessar essa referência e vê a lista b como contendo uma lista

```
Python 3.3
1 a = [1, 2, 3]
2 b = [0] * 7
→ 3 b[2] = a
→ 4 a[1] = 3
5 print(b)
```

Figura 18.7: Quarto Passo, alterar a

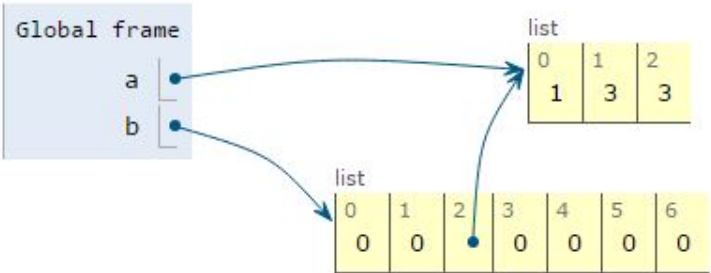


Figura 18.8: Agora quando alteramos a, o efeito é sentido em b.

[0, 0, [1, 3, 3], 0, 0, 0, 0]

Figura 18.9: Quinto passo, imprimir b, vemos que a mudança em a se reflete em b.

19

Coleções e Sequências

Vários tipos de Python, tanto *built-in* quanto pertencentes a módulos são *containers*, ou coleções, isto é, guardam dentro de si um número indefido de elementos.

Quando esses elementos são mantidos em ordem, dizemos que é uma sequência.

Quando podemos alterar um elemento sem alterar a identidade da coleção, dizemos que é mutável. Quando isso é impossível, dizemos que é imutável.

- Sequências
 - `list`, sequência mutável de qualquer coisa
 - `tuple`, sequência imutável de qualquer coisa
 - `str`, sequência imutável de caracteres. Ou, mais precisamente *Unicode code points*
 - `range`, sequência imutável de números
- Coleções não ordenadas
 - `set`, coleção de itens não repetidos (não-ordenada)
 - `dict`, coleção não ordenada de itens indexados por chaves. As chaves devem ser de tipos imutáveis.

Todas as sequências atendem as funções estabelecidas na Tabela 19.1

Função ou Operador	Utilidade
<code>x in s</code>	True se <code>x</code> é um item de <code>s</code> , False caso contrário
<code>x not in s</code>	True se <code>x</code> não é um item de <code>s</code> , False caso contrário
<code>s + t</code>	concatenação das sequências
<code>s*n</code> ou <code>n*s</code>	concatenar <code>s</code> consigo <code>n</code> vezes
<code>s[i]</code>	<code>i</code> -ésimo elemento de <code>s</code>
<code>s[i:j]</code>	fatia de <code>s</code> do elemento <code>i</code> ao <code>j</code>
<code>s[i:j:k]</code>	fatia de <code>s</code> do elemento <code>i</code> ao <code>j</code> , pulando <code>k</code>
<code>len(s)</code>	quantidade de elementos em <code>s</code>
<code>min(s)</code>	o menor de todos os elementos em <code>s</code>
<code>max(s)</code>	o maior de todos os elementos em <code>s</code>
<code>s.index(x, i, j)</code>	posição do elemento <code>x</code> em <code>s[i:j]</code>
<code>s.count(x)</code>	quantas vezes <code>x</code> aparece em <code>s</code>

Tabela 19.1: Funções e operadores aplicáveis a sequências `s` e `t`, item `x` e inteiros `i`, `j` e `cfk`

19.1 *Programas que tratam sequências*

Como os operadores da Tabela 19.1 servem a todas as sequências, podemos fazer funções que funcionam para qualquer tipo de sequência.

20

Repetindo instruções com for

20.1 Iteradores de Sequências

Apesar da generalidade do comando `while`, programadores Python geralmente preferem o comando similar `for`, que permite iterar por sequências ou qualquer outro tipo que forneça iteradores.

A seguir, um exemplo de um `for` muito simples.

```
for cor in [ "Vermelho", "Azul", "Verde" ]:  
    print(cor)
```

Que gera o resultado:

Chamamos a variável `cor` de um iterador. Ele vai receber cada valor encontrado dentro da lista, na ordem em que estão na lista¹.

Isso pode ser feito também com tuplas.

```
for cor in ( "Vermelho", "Azul", "Verde" ):  
    print(cor)
```

Que gera o resultado:

E se usado com uma string, funcionará com seus caracteres:

```
for letra in "abc":  
    print(letra)
```

Que gera o resultado:

Em Python, sempre devemos preferir soluções com `for` sobre soluções com `while`. Elas são mais elegantes e menos sensíveis a erros.

20.2 Usando Ranges

Muitas vezes usamos o `for` com uma lista de números gerada automaticamente pela função `range`. Essa função gera, a cada chamada, um novo número em uma sequência determinada por seus parâmetros, da seguinte forma:

- `range(n)`, como em `range(10)`, gera todos os números entre 0 e `n-1`, nesse exemplo de 0 a 9.

¹ Esse comportamento se repetirá para todas as sequências ordenadas, mas não acontece para dicionários, que não tem ordem garantida

- `range(i, f)`, como em `range(3, 7)`, gera todos os números entre `i` e `f-1`, nesse exemplo de 3 a 6
- `range(i, f, p)`, gera todos os números entre `i` e `f-1` pulando com o passo `p`. Se for `range(3, 9, 2)` gera os números 3, 5 e 7

Em Python 3, `range` é um tipo gerador de números. Para obter a lista dos números então precisamos usar a função `list`, como em:

20.3 *for versus while*

O uso de `range` permite que muitos problemas que são resolvidos com `while` e variáveis numéricas sejam substituídos. Isso tem a vantagem de não precisarmos inicializar a variável de contagem ou somar 1 a ela.

Por exemplo, somando todos os números de 1 a 10, resolvidos de ambos os jeitos.

Usando `while`:

```
i = 0
while i <= 10:
    print(i)
    i = i + 1
```

Usando `for`:

```
for i in range(1, 11):
    print(i)
```

Duas linhas a menos, duas chances a menos de errar!

20.4 *Problemas com For*

Exercício 22. Usando a instrução `for`, faça uma função `elimina(s, c)` que elimine todos os caracteres `c` de uma string `s`, sendo que por default deve eliminar espaços. Veja o Exercício 19.

```
def elimina(s, c):
    r = ""
    for letra in s:
        if letra != c:
            r = r + letra
    return r
```

20.5 *For e Dicionários*

No caso de dicionários, o iterador do `for` retorna uma chave e não o par. Não há garantia de ordem.

O programa:

```
d = { 1 : "Ovo" , "Banana" : 32, 12 : [2, 4, 5]}
for k in d:
    print("Chave:", k, "valor=>", d[k])
```

Resulta em:

20.6 Problemas com *for* e Dicionários

Exercício 23. Faça uma função que receba um dicionário cuja as chaves são nomes e os valores são uma lista de números e retorne um dicionário onde a chave são os mesmos nomes e os valores a soma dos números.

```
1 def soma(d):  
2     r = {}  
3     for k in d:  
4         r[k] = sum(d[k])  
5     return r
```

Atenção para o fato que não precisamos nos preocupar, nesse problema, com o tipo da chave, já que a instrução `for` resolve esse problema pegando todas as chaves.

Parte II

Conceitos Avançados

Nessa Parte ficarão alguns conceitos avançados.

Tuplas Nomeadas - namedtuple

Tuplas nomeadas são uma outra classe, que permite construir uma tupla onde seus elementos podem ser referenciados por nomes.

Para criar uma tupla nomeada precisamos criar a classe específica, que será subclasse de `tuple`, com o comando `collections.namedtuple(typename, field_names, verbose=False, rename=False)`, onde `typename` é uma string que define o nome da tupla, `field_names` é uma lista de string com o nome dos campos,

Bibliografia

Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. ACM Comput. Surv., 17(4):471–523, December 1985. ISSN 0360-0300. DOI: 10.1145/6041.6042. URL <http://doi.acm.org/10.1145/6041.6042>.

Roger Pressman. Software Engineering: A Practitioner's Approach. McGraw-Hill, Inc., New York, NY, USA, 7 edition, 2010. ISBN 0073375977, 9780073375977.

Índice

- [/, 39](#)
- [\(\), 33](#)
- [*, 33](#)
- [+, 33](#)
- [-, 33](#)
- [/, 33](#)
- [//, 33](#)
- [=, 47](#)
- [%, 33](#)
- [**kwargs, 64](#)
- [*args, 63](#)
- [None, 61](#)
- [def, 61](#)
- [id, 49](#)
- [return, 61](#)

- [abs\(x\), 37, 53](#)
- [acoplamento, 52](#)
- [algoritmo, 20](#)
- [Anaconda, 27](#)
- [and, 44](#)
- [Android, 26](#)
- [anotações, 67](#)
- [append, 100](#)
- [append:list, 100](#)
- [argumentos](#)
 - [com nome, 54](#)
 - [default, 53](#)
- [argumentos:nomeados indefinidos, 64](#)
- [argumentos:quantidade indefinida, 63](#)
- [atribuição, 47](#)
- [atribuição:dupla, 68](#)
- [atribuição:múltipla, 48](#)

- [base:binária, 33](#)
- [base:hexadecimal, 33](#)
- [base:octal, 33](#)
- [biblioteca, 55](#)
- [binário, 33](#)
- [bits na mantissa, 35](#)
- [bool, 44](#)
- [Booleanos, 44](#)

- [calculadora, 29](#)
- [coesão, 51](#)
- [comentários, 30](#)
- [complex, 39, 55](#)
- [complex\(x\), 37](#)
- [Complexos, 39](#)
- [computador:cpu, 17](#)
- [computador:dispositivos de entrada e saída, 19](#)
- [computador:memória, 19](#)
- [computador:partes, 17](#)
- [computador:virtual, 20](#)
- [concatenando strings, 39](#)
- [constante, 47](#)
- [conversão: entre tipos numéricos, 45](#)
- [count, 58, 100](#)
- [count:list, 100](#)
- [cpu, 17](#)

- [default, 53](#)
- [dispositivos de entrada e saída, 19](#)
- [divisão, 33](#)
 - [inteira, 33](#)
- [divmod, 68](#)
- [dupla atribuição, 68](#)

- [Eclipse, 27](#)
- [epsilon, 35](#)
- [escopo, 64](#)
- [escopo dinâmico, 66](#)
- [extend, 100](#)
- [extend:list, 100](#)

- [False, 44](#)
- [fatias de strings, 41](#)
- [fatorial, 81](#)
- [Fibonacci, 81](#)
- [float, 35](#)
- [float\(x\), 37](#)
- [for, 115](#)
- [for:dicionários, 116](#)
- [for:strings, 115](#)
- [função:string, 58](#)
- [função:uso, 53](#)
- [funções](#)
 - [matemáticas incorporadas, 37](#)
- [Funções incorporadas para números, 37](#)

- [hexadecimais, 33](#)
- [história de Python, 21](#)

- [Ideone, 26](#)
- [IDLE, 26](#)
- [imprimir:outras bases, 33](#)
- [in, 94, 99, 105](#)
- [in:listas, 99](#)
- [in:set, 105](#)
- [in:tuplas, 94](#)
- [indentação, 61](#)
- [index, 58, 100](#)
- [index:list, 100](#)
- [index:str, 58](#)
- [indexando strings, 40](#)
- [input, 70](#)
- [insert, 100](#)
- [insert:list, 100](#)
- [int, 32](#)
- [int\(x\), 37](#)
- [inteiros, 32](#)
- [iOS, 26](#)

- [j, 39](#)

- [laços de repetição, 85, 115](#)
- [len](#)
 - [strings, 40](#)
- [len:str, 58](#)
- [library, 55](#)
- [license, 4](#)
- [Linux, 25](#)
- [list\(x\), 91, 95](#)
- [list:append, 100](#)
- [list:count, 100](#)
- [list:extend, 100](#)
- [list:index, 100](#)
- [list:insert, 100](#)
- [list:pop, 100](#)
- [list:remove, 100](#)
- [list:reverse, 100](#)

- list:sort, 100
- listas:in, 99
- literal, 47
- loop, 85, 115
- lower, 58
- módulo
 - operador, 33
- módulos, 55
 - complex, 55
 - math, 55
- MacOS, 25
- maior positivo, 35
- maior quantidade de algoritmos na
 - significativos, 35
- math, 55
- max(x,y,z,...), 37
- memória, 19
- memória:posição, 49
- memória:principal, 19
- memória:secundária, 19
- menor positivo, 35
- min(x,y,z,...), 37
- modules, 55
- números, 32
- números:inteiros, 32
- números:reais, 35
- None, 45
- not, 44
- not in, ver in
- notação científica, 36
- octal, 33
- online Python, 26
- operadores, 33
 - (), 33
 - *, 33
 - +, 33
 - , 33
 - /, 33
 - //, 33
 - %, 33
 - divisão
 - inteira, 33
 - módulo, 33
 - resto, 33
- operadores lógicos, 44
- or, 44
- ordem de precedência, 45
- packages, 57
- pacotes, 57
- pop, 100
- pop:list, 100
- posição de memória, 49
- precedência, 45
- precedência:operadores lógicos, 44
- print, 69
- programação:dificuldade, 21
- Pytho:o que é, 22
- Pytho:site, 25
- Python Tutor, 26
- Python:história, 21
- Python:obter, 25
- Python:online, 26
- Python:versão, 22
- range, 115
- reais, 35
- recursão, 81
- remove, 100
- remove:list, 100
- resto, 33
- reverse, 100
- reverse:list, 100
- set:in, 105
- site de Python, 25
- sort, 100
- sort:list, 100
- Spyder, 27
- standard library, 55
- str, 39
- str:count, 58
- str:index, 58
- str:lower, 58
- str:upper, 58
- string, 39
- string:função, 58
- strings
 - [], 40
 - concatenação, 39
 - fatias, 41
 - indexando, 40
 - len, 40
 - tamanho de uma..., 40
- sys.float_info, 35
- tamanho de uma string, 40
- Tipo, 31
- Tipo de Dados, 31
- True, 44
- tuplas:in, 94
- Ubuntu, 25
- unidade de processamento central,
 - 17
- upper, 58
- valor verdade, 45
- variáveis, 47
- variável, 47
- variável:escopo, 64
- while, 85
- Windows, 25
- yipos numéricos, 32

Lista de tarefas pendentes