

Técnicas de compressão para introdução à computação

PDPD 2016

Autor: João Miguel Moreno Ferreira Lima

Orientador: Vinicius Cifú Lopes

Introdução

Este trabalho dissertará sobre a compressão de dados digitais, explicando quais serão as ferramentas escolhidas e como serão utilizadas. Será feita uma breve explicação sobre o que é a linguagem de programação *Python* - escolhida para desenvolver os algoritmos a serem implementados - e sua relação com a computação científica, bem como será construído um manual de instalação e configuração do *Python* nos ambientes Windows e GNU/Linux. Em seguida, inspirado em Coelho (2007), serão elucidadas as principais estruturas de programação e de dados relacionadas ao *Python*: o que são e como as utilizar; e como criar um novo programa e realizar sua execução.

Este capítulo tem o objetivo principal de permitir que qualquer leitor deste material seja capaz de reproduzir, por conta própria, os estudos aqui realizados.

Computação Científica

Computação científica é a aplicação das técnicas computacionais digitais no desenvolvimento da ciência, seja qual for a área do conhecimento a

ela atrelado. Contudo, aplicar a computação científica em um projeto de pesquisa não é algo trivial pois exige um nível minimamente razoável de conhecimento, tanto de programação quanto da área da pesquisa.

Com isso, algumas empresas surgiram no mercado de software oferecendo aplicações genéricas direcionadas à computação científica, como é o caso do MATLAB™, do Mathematica™, entre outros. Ainda assim, muitas áreas da ciência acabam não sendo contempladas com o desenvolvimento proprietário de softwares, forçando a própria comunidade científica a desenvolver soluções que atendam suas demandas.

Dessa forma os cientistas precisaram, por meios próprios, desenvolver soluções computacionais ainda que possuam poucos conhecimentos na área de programação e pouco tempo para dividir com a atividade de pesquisa principal. Contudo, uma grande barreira no desenvolvimento de softwares por cientistas é a dificuldade com as linguagens de programação, que geralmente não são triviais, demandam um grande tempo de aprendizado e possuem regras sintáticas complexas, como são os casos das linguagens C, C++, Java, entre outras.

Surge como alternativa facilitadora para esses cientistas a linguagem de programação *Python* que, ao contrário das tradicionais linguagens de programação como as citadas acima, possui uma curva de aprendizagem muito menor e com regras sintáticas extremamente mais simples. Isto é, com menos linhas de código e com menos tempo de dedicação, um mesmo programa que seria feito em C, C++ ou Java é feito em *Python*.

Sobre o *Python*

É uma linguagem de programação *open-source*, de alto nível, interpretada e interativa.

Open-source, segundo *The Open Source Initiative* – Iniciativa Código Aberto, em português – é um movimento global que promove não somente a abertura do código fonte pelo seu autor a qualquer um que tenha interesse em estudá-lo, mas também define alguns critérios de distribuição do *software*. Dentre esses critérios destacam-se a livre distribuição desse *software* juntamente com seu código fonte aberto e a liberdade de realizar modificações ou de desenvolver derivações do *software* livre original e que também devem seguir estes critérios de distribuição.

Além disso, por ser de alto nível e interativa é uma linguagem que possui uma estrutura sintática, isto é, regras de escrita muito simples e fáceis de compreender. Isto significa que a curva de aprendizado de qualquer programador ou não programador com o *Python* tende a ser muito pequena, dando bons resultados com muito menos esforço que linguagens como *C*, *C#*, *Java*, etc. E o fato de ser interativa, o que é uma grande novidade em relação às demais linguagens populares, proporciona que o programador consiga executar em tempo real pequenos blocos de código diretamente no interpretador do *python*, facilitando com que ele assimile as regras sintáticas com a lógica que pretende utilizar em seu *software*. Isso reduz em muito o tempo perdido para aprender um novo comando e assimilar a forma como utilizá-lo em seu código.

Linguagem ser interpretada, como é o *python*, significa que o interpretador realizará a tradução do código fonte original para linguagem de

máquina, instrução por instrução, ao mesmo tempo em que executa o programa. Logo, os erros sintáticos existentes no código só serão percebidos em tempo de execução do código, ou seja, no momento que o interpretador realizar a leitura da instrução que contiver o erro. Outro ponto a se destacar das linguagens interpretadas é que elas possuem uma velocidade de processamento menor que as linguagens compiladas. Então, quando um programador desenvolve uma aplicação é preciso atentar-se a essa questão também: performance operacional do programa.

Instalação e configuração do *Python*

O início prático deste capítulo dá-se aqui, com a instalação e configuração do python nos sistemas operacionais Windows e GNU/Linux. Para tanto, atente-se ao sistema operacional instalado em sua máquina e siga os passos correspondentes abaixo.

Sistema Operacional Windows

Para instalar o *Python* no *Windows*, clique no link <https://www.python.org/ftp/python/2.7.13/python-2.7.13.amd64.msi>. Note que a versão a ser instalada pertence à série 2.7x. Em seguida, execute o instalador e avance as etapas de instalação até o final (dê preferência para a instalação padrão) conforme as imagens abaixo:



Figura 1: Instalação Python - Instalar para todos os usuários

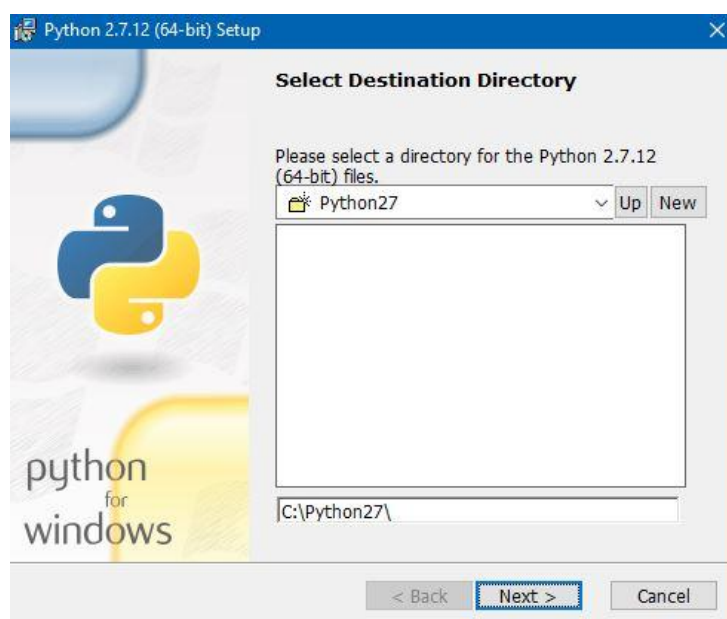


Figura 2: Instalação Python - Instalar no diretório padrão



Figura 3: Instalação Python - Instalar as ferramentas padrão

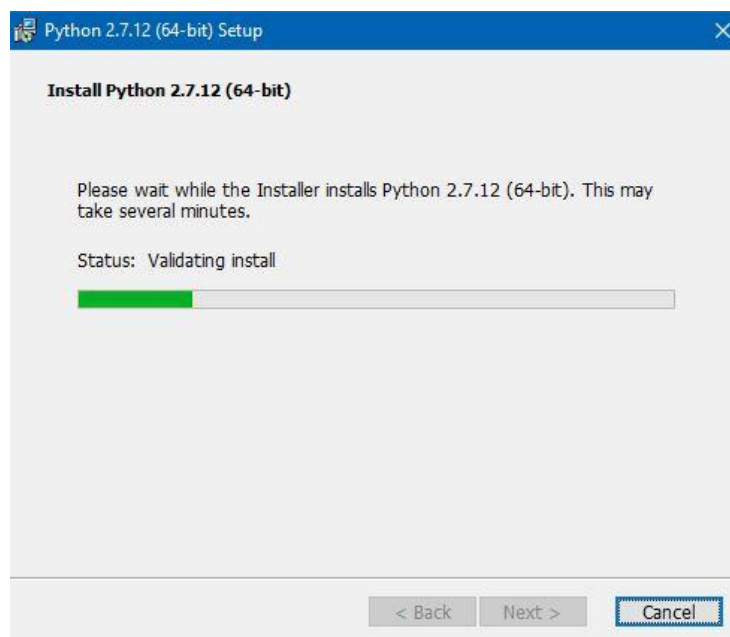


Figura 4: Instalação Python - Aguardar a instalação ser completada



Figura 5: Instalação Python - Finalizar a instalação

Ao final, depois de instalado, navegue até o menu iniciar e encontre o aplicativo “*Python (command line)*” e abra-o:

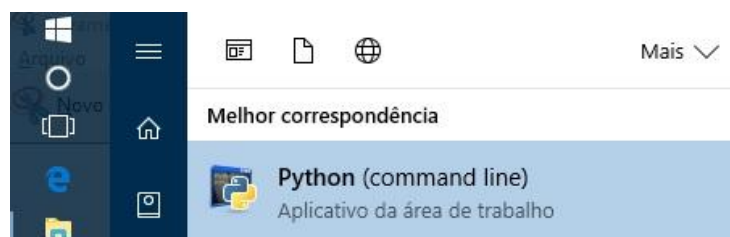


Figura 6: Iniciando o Python em Windows

Ao abrir, aparecerá a janela de linha de comando do Python, como na imagem abaixo:

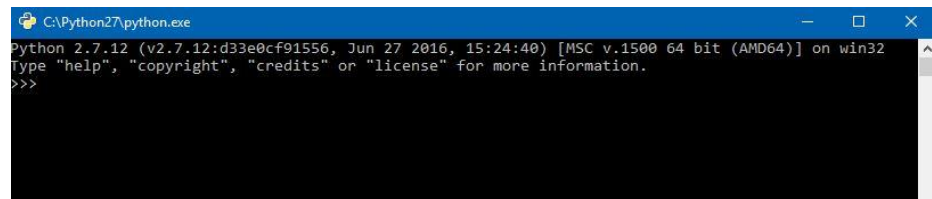


Figura 7: Janela do interpretador de comandos do Python

Note que a versão do Python aparecerá na primeira linha abaixo do comando digitado. Note também que a última linha possui três símbolos iguais em sequência ">>>", isto é, sempre após esse símbolo é que serão digitados os comandos de programação do python.

Outra etapa a ser feita após a instalação é a configuração da variável de ambiente do python no Windows para que ele reconheça os comandos do python digitados no prompt de comando.

Para isso, abra o menu iniciar e pesquise o aplicativo Sistema, como na imagem abaixo:

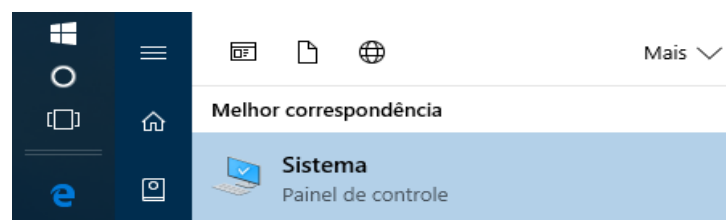


Figura 8: Acessar o aplicativo Sistema do Windows

Em Seguida, na janela Sistema, clique em Configurações Avançadas do Sistema:

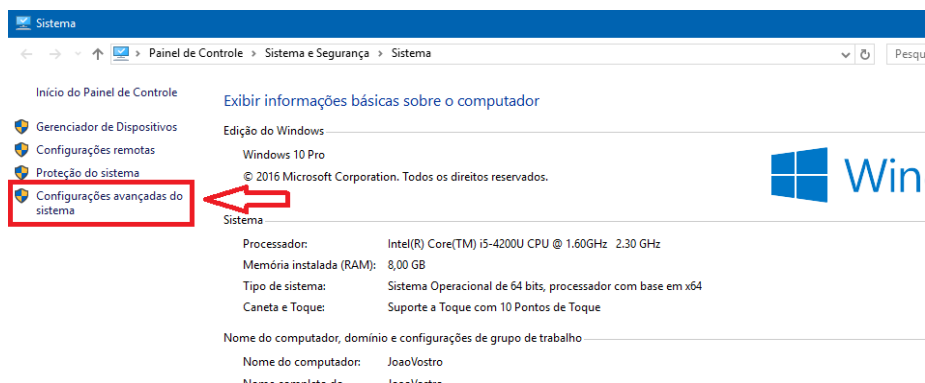


Figura 9: Janela de sistema do Windows

Na janela Propriedades do Sistema, que abrir, clique no botão Variáveis de ambiente:

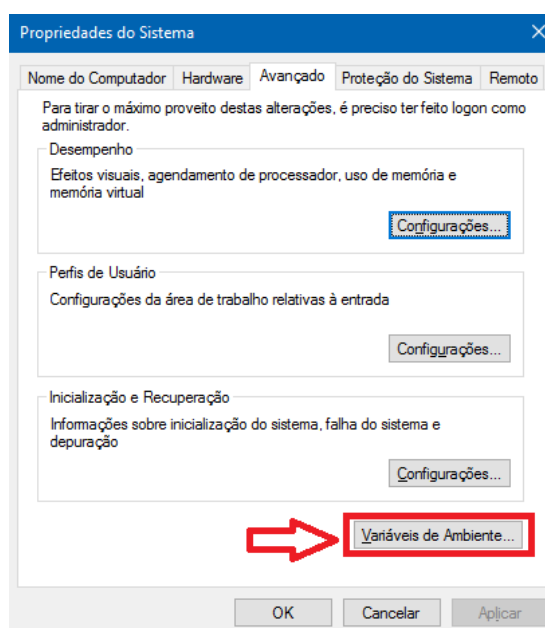


Figura 10: Janela de Propriedades do Sistema

Na nova janela de Variáveis de Ambiente que surgir encontre na janela de navegação Variáveis do Sistema a variável Path e clique em editar:

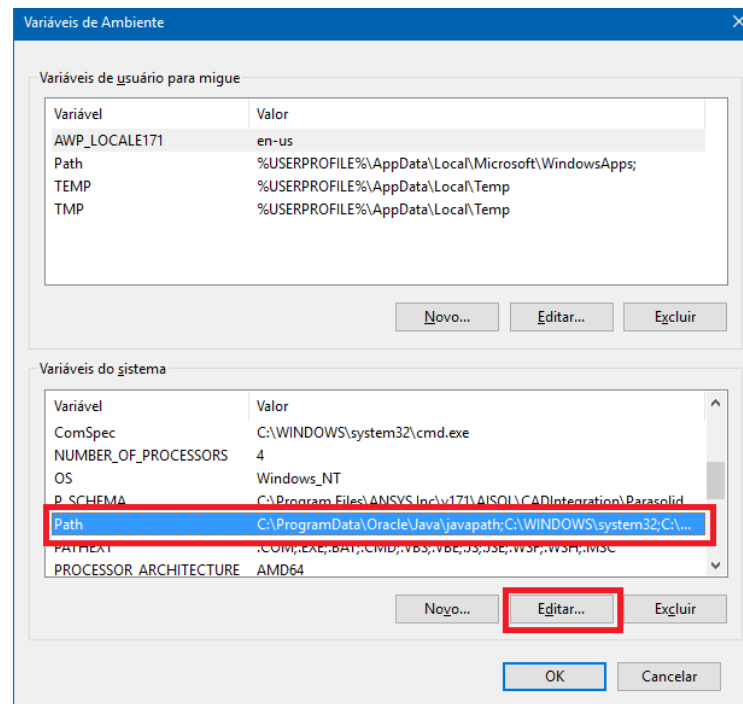


Figura 11: Janela de Variáveis de Ambiente

Aparecerá a janela Editar a Variável de Ambiente e então clique no botão novo e insira o seguinte caminho `C:\python27`:

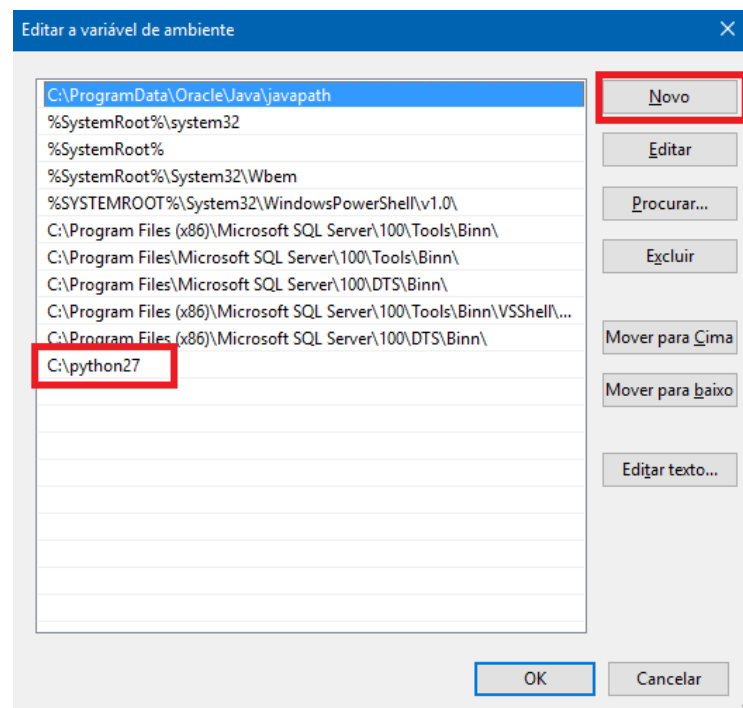


Figura 12: Janela Editar Variável de Ambiente

Por fim clique em *OK* e feche as janelas.

Para ter a certeza de que o *python* funciona no *Windows* vá ao menu iniciar e procure o aplicativo *prompt* de comando. Ao abri-lo aparecerá uma tela preta:

Como na imagem abaixo, digite *python*:



Figura 13: Janela do Prompt de comando

Observe se aparecerá a versão do *python* instalada e os três símbolos em sequência “>>>”, como na imagem:

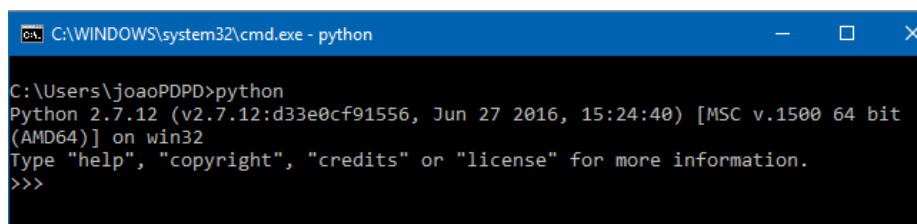
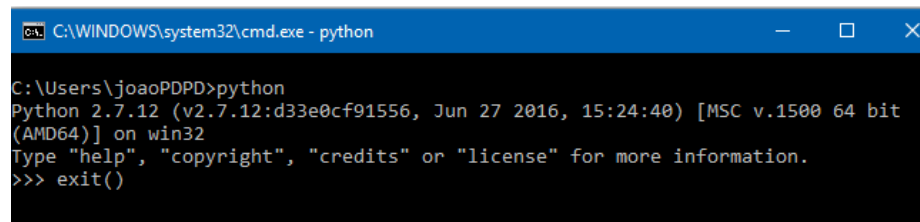


Figura 14: Acessando o Python pelo Prompt de comando

Com isso, o *python* então estará instalado e configurado para uso no *Windows*.

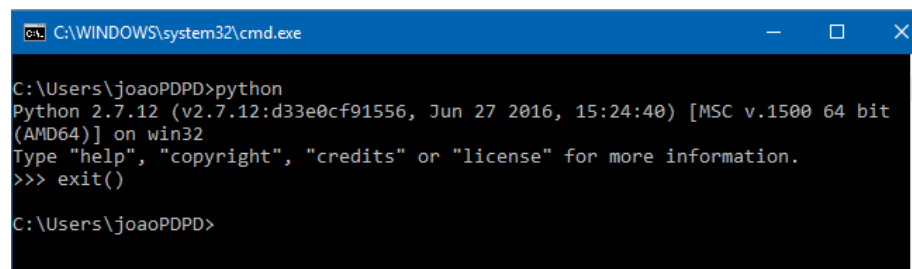
Uma vez com a janela *Prompt de comando – Python* aberta, para sair basta digitar, logo após os três símbolos “>>>”, o comando *exit()* e, em seguida, clicar no botão de fechar a janela do *prompt* de comando.



```
C:\WINDOWS\system32\cmd.exe - python
C:\Users\joaoPDPD>python
Python 2.7.12 (v2.7.12:d33e0cf91556, Jun 27 2016, 15:24:40) [MSC v.1500 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
>>>
```

Figura 15: Encerrando o Python

Note que após o *Python* ser encerrado corretamente os três símbolos sequenciais “>>>” não estarão mais disponíveis e a janela do terminal poderá ser fechada corretamente.



```
C:\WINDOWS\system32\cmd.exe
C:\Users\joaoPDPD>python
Python 2.7.12 (v2.7.12:d33e0cf91556, Jun 27 2016, 15:24:40) [MSC v.1500 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
C:\Users\joaoPDPD>
```

Figura 16: Python encerrado

Sistema Operacional *Linux*

Este tópico abordará a instalação e configuração do *Python* no sistema operacional *Linux* baseado na distribuição *Debian*, como o *Ubuntu* e o *Linux Mint*. As distribuições *Linux* baseadas no *Debian* possuem o mesmo gerenciador de pacotes - aplicação interna do *Linux* utilizada para instalar softwares através de linhas de comando, o APT, ou “*Advanced Packaging Tool*”.

Com isso, para realizar a instalação do *Python* no *Linux*, siga as seguintes etapas:

Abra o terminal de comando do *Linux*:

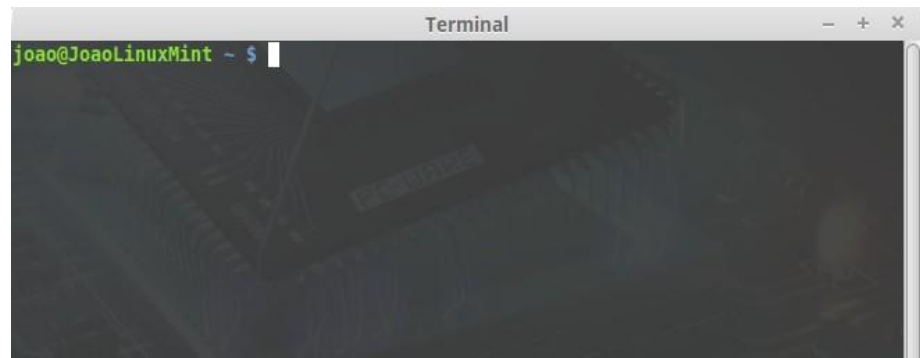


Figura 17: Abrir o terminal de comandos no Linux

Após o símbolo \$ digite o comando `sudo apt-get install python` e, em seguida, digite a senha do seu usuário de máquina local quando for solicitado. A instalação iniciará assim que a senha for digitada:

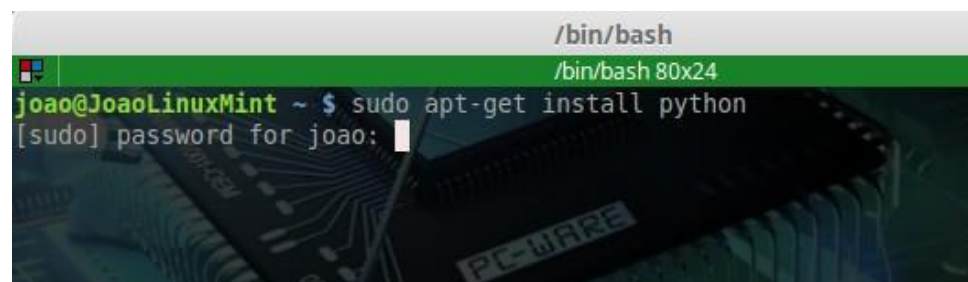


Figura 18: Comando para instalar o python no Linux

Ao final, para verificar se o *python* fora instalado corretamente, digite apenas o comando `python`, como na imagem:

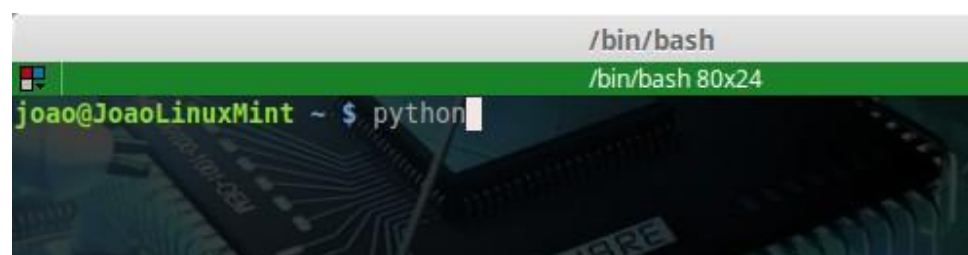


Figura 19: Comando de inicialização do Python no Linux

Deverá aparecer uma imagem semelhante a essa:

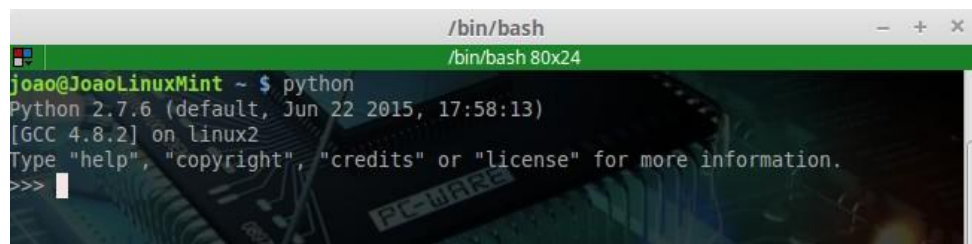
A terminal window titled '/bin/bash' with a green header bar. The prompt is 'joao@JoaoLinuxMint ~ \$'. The command 'python' has been entered, and the output shows 'Python 2.7.6 (default, Jun 22 2015, 17:58:13)' and '[GCC 4.8.2] on linux2'. Below this, it says 'Type "help", "copyright", "credits" or "license" for more information.' and the prompt '>>>' is shown with a cursor.

Figura 20: Python inicializado no Linux

Note que a versão do *Python* aparecerá na primeira linha abaixo do comando digitado. Note também que a última linha possui três símbolos iguais em sequência “>>>”, isto é, sempre após esse símbolo é que serão digitados os comandos de programação do *python*.

Com isso, o *python* então estará instalado, no *Linux*.

Uma vez com a janela do terminal aberta e com o *Python* inicializado, para sair basta digitar, logo após os três símbolos “>>>”, o comando *exit()* e, em seguida, clicar no botão de fechar a janela do *prompt* de comando, como nas imagens a seguir:

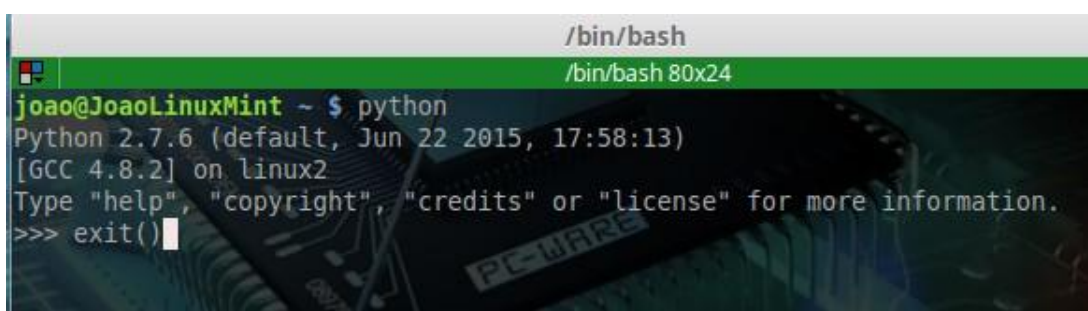
A terminal window titled '/bin/bash' with a green header bar. The prompt is 'joao@JoaoLinuxMint ~ \$'. The command 'python' has been entered, and the output shows 'Python 2.7.6 (default, Jun 22 2015, 17:58:13)' and '[GCC 4.8.2] on linux2'. Below this, it says 'Type "help", "copyright", "credits" or "license" for more information.' and the prompt '>>>' is shown. The command 'exit()' has been entered, and the cursor is at the end of the line.

Figura 21: Encerrando o Python

Note que após o *Python* ser encerrado corretamente os três símbolos sequenciais “>>>” não estarão mais disponíveis e a janela do terminal poderá ser fechada corretamente.

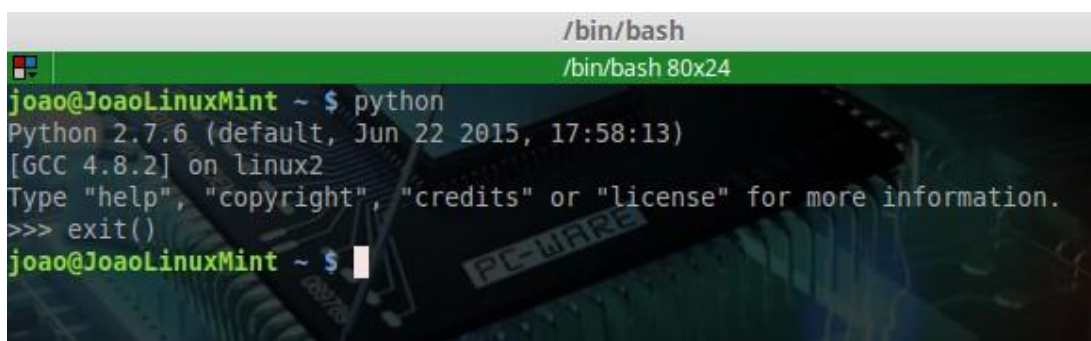
A screenshot of a terminal window titled "/bin/bash" and "/bin/bash 80x24". The prompt is "joao@JoaoLinuxMint ~ \$". The user enters "python", which starts the Python 2.7.6 interpreter. The interpreter displays its version and environment information: "Python 2.7.6 (default, Jun 22 2015, 17:58:13) [GCC 4.8.2] on linux2". It then prompts the user with "Type 'help', 'copyright', 'credits' or 'license' for more information." The user enters ">>> exit()", which returns the prompt to the shell: "joao@JoaoLinuxMint ~ \$". The terminal background has a dark, abstract pattern.

Figura 22: python encerrado

Elementos Básicos da Linguagem *Python*

Serão abordados nesse tópico alguns dos elementos funcionais e estruturais mais relevantes do *python* e que serão amplamente utilizados no desenvolvimento dessa pesquisa. O conhecimento desse tópico é de extrema relevância para que a leitura e a compreensão do trabalho como um todo seja efetiva e possível de ser reproduzida.

A seguir serão descritos em detalhes como funcionam os modos de execução de comandos no *python*, o que é o interpretador de código *python*, a declaração de variáveis e como o *python* atribui um tipo de dado a essas variáveis, algumas estruturas de dados e os principais controles de fluxo de execução de um código fonte.

Execução Interativa e Execução via *script*

O Python foi concebido de tal forma que permite ao programador executar comandos de forma interativa. Em outras palavras, o programador pode testar o próprio código enquanto o escreve. É uma maneira útil e eficiente de se testar tanto algumas combinações de operações como a lógica de um pequeno trecho de código.

Todavia, as construções de software mais elaboradas são feitas via arquivos *script*, que são os arquivos que contém todas as linhas de código necessárias para a execução do programa. Além disso, a execução via *script* permite que o programador salve e execute o código mais de uma vez e que o transporte de um dispositivo para outro.

O interpretador *Python*

O computador, mais especificamente o processador, realiza instruções e operações escritas em uma linguagem muito específica e que é praticamente ilegível pelo ser humano. É conhecida como linguagem de máquina e linguagem de baixo nível. Consequentemente, precisa haver um método que possibilite a escrita, leitura e compreensão de um código fonte por qualquer programador humano. Com isso surgiram as linguagens de programação de alto nível, que abstraem a complexidade das linguagens de baixo nível a tal ponto que qualquer pessoa minimamente preparada possa ler e ter uma razoável compreensão do que aquele código faz.

Contudo, como dito anteriormente, um código fonte em alto nível não pode ser compreendido pela máquina e, por isso, faz-se necessário a tradução de um código alto nível para um código em linguagem de máquina, de baixo

nível. Um dos métodos de tradução desse código é a interpretação, que será explicada a seguir.

Segundo Barreto (2001), o interpretador nada mais é que um programa com a capacidade de ler um código em uma linguagem de alto nível e traduzi-lo para uma linguagem de baixo nível. Essa leitura é realizada instrução por instrução do código fonte original. A cada instrução lida, o interpretador verifica se há erros de escrita (erros sintáticos) e, em seguida, converte-a para uma instrução de linguagem de máquina e, então, envia ao computador para que execute a instrução. Esse processo é repetido com todas as instruções que compõem o programa a ser executado.

O interpretador *python* é responsável, portanto, por realizar a tradução de um código escrito em linguagem *python* para a linguagem de máquina, conforme o método descrito acima.

Comportamento e declaração de variáveis

Antes de mencionar a declaração de variáveis é necessário introduzir a ideia de tipo de dado. Tipo de dado nada mais é que a natureza do dado que uma variável armazena, por exemplo, dado do tipo inteiro, decimal, caractere, cadeia de caracteres, entre outros.

No *python* a atribuição de tipo de dado à uma variável é automática, ou seja, ocorre quando o interpretador lê a instrução de declaração de variável e o valor a ela associado.

A declaração de uma variável qualquer dentro de um código *python* é simples e direta: basta escrever um nome que se deseja para a variável

seguido de um símbolo de igualdade e, por fim, seguido do valor que a variável receberá.

```
<nome da variável> = <valor da variável>
```

Exemplos de declaração de variável:

```
a = 2
```

```
idade = 15
```

```
ano = 2016
```

No Python, a declaração de valores decimais é feita com a utilização do ponto “.” como separador da parte inteira e da parte decimal, conforme exemplos abaixo:

```
x = 3.5
```

```
preco = 2.99
```

```
gasolina = 3.49
```

Outra particularidade importante de se salientar é a declaração de variável cujo valor é um objeto literal, ou seja, um caractere ou uma sequência de caracteres. Neste caso, o valor a ser atribuído à variável deve estar delimitado ou por aspas ou por apóstrofo, como nos exemplos a seguir:

```
letra = "a"
```

```
nome = "Joao Pedro"
```

```
vogal = 'o'
```

```
endereco = 'Rua Utinga, 205'
```

```
nome = "Pedro"
```

Uma vez declarada a variável, atribuído um valor a ela e executado o código, essa variável poderá ser manipulada somente com operações compatíveis com seu tipo. Isto significa, por exemplo, que a variável **letra** = “a” admite apenas operações compatíveis com o tipo de dado caractere, ou seja, não há a possibilidade de se realizar uma operação aritmética com a variável letra pois esta não é uma variável do tipo inteiro ou decimal. Isso ocorre em todos os tipos de dados.

Operações e atribuições de valores

O Python reconhece algumas operações matemáticas e não matemáticas como a soma, subtração, multiplicação, divisão e a concatenação, que serão exemplificadas a seguir utilizando o *Python* interativo, uma vez que são instruções extremamente básicas que não exigem a construção de arquivos de código fonte.

Soma (+):

```
>>> 2 + 2
```

```
4
```

Subtração (-):

```
>>> 4 - 2
```

```
2
```

Multiplicação (*):

```
>>> 3 * 2
```

```
6
```

Divisão (/):

```
>>>7 / 2
```

```
2
```

Neste caso da divisão, o *Python* ao dividir 7 por 2 realiza a divisão entre dois números inteiros resultando em um número inteiro. Por isso o resultado obtido não é o esperado, mas um arredondamento para o inteiro mais próximo. Para obter um resultado com valor decimal é necessário que pelo menos uma das partes da divisão seja um número decimal e isso se faz colocando o ponto “.” depois de um dos números, ou de ambos, como nos exemplos a seguir:

```
>>>7. / 2
```

```
3.5
```

```
>>>7 / 2.
```

```
3.5
```

```
>>>7. / 2.
```

```
3.5
```

Essa lógica para obter resultados decimais com números inteiros é a mesma para a soma, subtração e multiplicação.

Há também outro ponto necessário de ser compreendido que é a acumulação de um valor em uma variável que já foi declarada e valorada. Por exemplo, uma variável denominada contadora (essa ideia de variável contadora é extremamente utilizada em qualquer programa) inicia com valor 0. Em certo momento do programa, esse valor 0 precisa ser incrementado com um novo valor repetidas vezes. A forma de representar essa variável é a seguinte:

```
contadora = contadora + i
```

Isto é, cada vez que essa instrução for executada o interpretador *Python* irá somar o atual valor da variável *contadora* com o valor da variável *i* e o resultado irá armazenar na própria variável *contadora*, substituindo o valor antigo.

Porém, há uma forma sintética de representar a mesma operação, cuja estrutura é:

```
contadora += i
```

Neste segundo caso, os símbolos `+=`, juntos, significam que a variável *contadora* vai somar à ela mesma o conteúdo de *i* e armazenar em si o resultado.

Por outro lado, conforme exposto logo acima, as variáveis que armazenam dados literais não permitem a realização de operações aritméticas, como a soma. Porém, quando se associa a simbologia do sinal de adição (+) a uma variável cujo conteúdo seja literal, o interpretador *Python* automaticamente reconhece a operação como uma concatenação (junção) de dados literais. Por exemplo, supondo as variáveis *nome* e *sobrenome*, como abaixo:

```
nome = "Pedro"
```

```
sobrenome = " Henrique"
```

Se for realizada a operação de concatenação, dada pelo uso do símbolo de +, temos como resultado a junção dos conteúdos das duas variáveis literais em um conteúdo literal único resultante, como no exemplo abaixo:

```
>>>nome + sobrenome
```

```
Pedro Henrique
```

Há o caso, ainda, em que desejamos acumular valor literal em uma mesma variável. Por exemplo, a variável *nome*, usada acima, possui o conteúdo literal *Pedro*. Há a possibilidade de se concatenar dados literais ao conteúdo dessa variável e o conteúdo resultante continuar sendo armazenado na própria variável *nome*, como no exemplo a seguir:

```
nome = "Pedro"

>>> nome = nome + " Henrique"

>>> nome

Pedro Henrique
```

Assim como na variável contadora, a variável *nome* foi utilizada como uma variável acumuladora de dados, porém literais desta vez. E também, como na variável contadora, podemos reduzir essa expressão para uma forma mais enxuta, como no exemplo abaixo:

```
nome = "Pedro"

>>> nome += " Henrique"

>>> nome

Pedro Henrique
```

Estruturas de Dados

De acordo com NIST (2004), estrutura de dados é um modo de organizar os dados de um programa a fim de se obter melhor performance

durante sua execução. As estruturas de dados podem ser criadas de diferentes maneiras de forma que atendam diferentes necessidades de manipulação desses dados, levando em conta a performance do algoritmo.

Alguns tipos de estruturas de dados em *python* são as listas, as tuplas, as cadeias de caracteres e os dicionários, conforme serão explicadas abaixo.

Listas

As listas são estruturas de dados que representam um conjunto de dados delimitados por colchetes. Cada dado é um elemento da lista e é armazenado ordenadamente. A declaração explícita de uma lista com n elementos é feita separando cada elemento do seu sucessor por meio de uma vírgula. Além disso, como seus elementos estão ordenados, há uma indexação na lista para identificar cada elemento. Essa indexação inicia-se em zero para representar o primeiro elemento e termina em $n - 1$ para representar o $n - \text{ésimo}$ elemento.

Por exemplo:

```
lista1 = [1, "casa", "c", -1]
```

```
lista2 = [1, 3, 5, 7]
```

```
lista3 = ["a", "b", "z", "w", "f"]
```

Na variável *lista3*, o índice 0 corresponde à letra a; o índice 1, à letra b; o índice 2, à letra z e assim por diante. Analogamente, isso vale para as variáveis *lista1* e *lista2*.

Tuplas

Em matemática, tuplas são sequências finitas de elementos ordenados. Quando se tem uma sequência com nenhum elemento, essa tupla é vazia. Quando possui um elemento, é chamada de monupla. Analogamente, para dois e três elementos é chamada, respectivamente, de dupla e tripla. Quando se tem n elementos, portanto, denota-se genericamente $n - upla$, em português. Contudo em inglês uma tupla genérica de n elementos é chamada de $n - tuple$. Logo, convencionou-se usar, na computação principalmente, apenas o termo tupla, derivado do inglês, para designar uma lista de elementos ordenados.

No *python*, as tuplas são conjuntos de elementos ordenados como as listas. Contudo, a diferença entre elas reside no fato de as tuplas serem imutáveis, isto é, uma vez criada, seu conteúdo é inalterável. Sua utilidade se dá no ganho de performance de processamento por ser mais rápida que a lista. Útil, portanto, quando não há a necessidade de alterar seus dados armazenados, servindo apenas de referência. E, bem como nas listas, seu índice inicial é o zero.

Por exemplo:

```
tupla1 = ("joao", 1, "f", -23.5)
```

```
tupla2 = ("a", "b", "z", "w")
```

```
tupla3 = ("casa", "carro", "computador")
```

Cadeias de caracteres ou *Strings*

São sequências encadeadas de caracteres individuais e são delimitadas por apóstrofo ou aspas duplas. Atente-se ao uso do termo *String*, oriundo da

língua inglesa, para designar uma cadeia de caracteres. Devido ao uso intensivo desse termo em detrimento do primeiro, qualquer leitor da área de computação se deparará praticamente apenas com o uso da expressão *String*. Seguindo essa tendência, esse trabalho também reproduzirá esse termo sempre que for necessário falar sobre cadeia de caracteres.

Por exemplo:

```
string1 = 'Compressao de dados'
```

```
string2 = "Compressao de dados"
```

A diferença entre as variáveis *string1* e *string2* está no uso das aspas ou do apóstrofo para delimitar uma cadeia de caracteres.

Dicionários

São conjuntos de elementos não ordenados e que a cada elemento nele inserido uma chave de acesso deve ser associada. É delimitado por chaves e para cada elemento a ser inserido há dois dados associados, que são a chave de acesso e o valor do elemento, separadas por dois pontos, conforme a estrutura a seguir:

```
<nome do dicionario> = {<valor da chave> : <valor do  
elemento>}
```

Por exemplo:

```
dicio1 = {1: "Pedro", 2: 345, "c": "a"}
```

```
dicio2 = {"chave1": 1, "chave2": 2, "chave3": 3.5}
```

Comentários no Código Fonte

Um recurso muito importante dentro de qualquer linguagem de programação é o uso de linhas ou blocos de texto que não são processados pela linguagem. São os chamados comentários e são utilizados para documentar e explicar como o código funciona. No *Python* os comentários são precedidos, a cada linha, pelo símbolo “#”.

Por exemplo:

```
#Atribuição do valor 15 à variável idade  
  
idade = 15
```

Este foi um exemplo com apenas uma linha de comentário. Outro exemplo, a seguir, com duas ou mais linhas de comentário escritas dentro do código fonte:

```
#O código abaixo atribui valores às variáveis valor1 e  
#valor2  
  
#Em seguida, atribui à variável resultado a soma entre  
#valor1 e valor2  
  
valor1 = 3.5  
  
valor2 = 2.0  
  
resultado = valor1 + valor2
```

Perceba que nos comentários o uso de acentuação é livre, pois o interpretador não executa comentário. Contudo, o código executável não deve possuir acentuação pois isso gera incompatibilidade e erros.

Indentação ou recuo

No python, a indentação pode ser definida como o recuo de uma linha de código em relação à sua margem à esquerda. Isso se faz necessário para criar uma maneira de se identificar hierarquia entre instruções sequenciais dentro de um código. Em outras palavras, as linhas de código que estiverem à mesma distância da margem possuem o mesmo nível hierárquico de execução do código, sendo independentes entre si. Em contrapartida, as instruções com maior recuo, ou seja, com maior distância em relação à margem esquerda, são instruções que estão subordinadas às instruções imediatamente com recuo menor. E assim sucessivamente.

Especificamente no python esse recuo pode ser obtido com pelo menos uma unidade de espaço ou uma unidade de tabulação, sendo mais recomendado que se use o espaço para indentação.

A indentação do código é de vital importância para os controles de fluxo explicados a seguir.

Controles de Fluxo

Quando um programa é interpretado ele é lido e traduzido sequencialmente, instrução a instrução, para uma linguagem de baixo nível. Contudo, há estruturas que geram exceções nesse fluxo linear de interpretação do código: são os chamados controles de fluxo. Abaixo veremos alguns mais relevantes, que são reconhecidos pelo *Python*, e que são universais a quaisquer linguagens de programação.

Controles condicionais (*If*, *elif* e *else*)

O controle condicional é uma estrutura que avalia uma determinada situação e, dependendo de seu estado momentâneo, fluxos de processamento diferentes podem ser tomados. Mais claramente, o que acontece é o seguinte: “se a situação é X, então faça isso, do contrário faça aquilo”.

Um primeiro caso do uso do condicional ocorre com a necessidade de apenas uma verificação de situação e seu respectivo bloco de código a ser executado caso essa situação seja satisfeita. Ou seja:

```
if variavel == valor

    #então executa este bloco de código
```

Neste caso, o condicional é utilizado apenas para executar um trecho de código caso uma determinada situação seja satisfeita, não importando o caso em que a condição não seja satisfeita.

Porém há casos em que é necessária a execução de um determinado código em situação adversa à condição principal. Para isso, há pelo menos duas estruturas possíveis:

Estrutura 1:

```
if variavel == valor1:

    #então excute este bloco de código

else:

    #Se a condição acima não foi satisfeita, então
    #execute este bloco de código
```

Nesse modelo de condicional, a lógica é basicamente a seguinte: “Se a situação é X, então faça isso, senão faça aquilo”. Há apenas uma condição

explícita e caso ela não seja satisfeita, um segundo bloco de código é executado.

Estrutura 2:

```
if variavel == valor1:

    #então excute este bloco de código

elif variavel == valor2:

    #então execute este bloco de código

else:

    #Se nenhuma condição acima foi satisfeita, então
    #execute este bloco de código
```

Nessa segunda estrutura a lógica é parecida com a da estrutura 1, com a diferença de que há mais de uma condição de verificação explícita. A lógica é a seguinte: “Se a situação é X, então faça isso. Senão, se a situação for Y, então faça esse outro. Do contrário, então faça aquilo”.

Uma observação extremamente importante é que em qualquer controle de fluxo a indentação é imprescindível para que o interpretador execute somente as linhas correspondentes a esse controle de fluxo. Por exemplo:

```
if valor == 2:

    resultado = valor + 5

    print resultado

valor = 0
```

Neste exemplo, as linhas “*resultado = valor + 5*” e “*print resultado*” fazem parte do condicional “*if valor == 2:*”, enquanto a linha “*valor = 0*” não. O interpretador do *Python* compreende isso pois o bloco de código dentro do condicional está indentado, como já explicado anteriormente, em relação ao condicional. O que não estiver indentado não faz parte do mesmo bloco de código, como é o caso da linha “*valor = 0*”.

Essa questão da indentação é extremamente importante em todos os controles de fluxo, tanto no condicional, quanto nos controles de iteração a serem explicados a seguir.

Controles de iteração

A iteração é o processo de se repetir quantas vezes forem necessárias um bloco de código, sem que tal bloco seja escrito repetidas vezes. Para tal, esse trecho de código que deve ser iterado é escrito apenas uma vez dentro de um controle de iteração que o repetirá segundo determinadas condições.

Alguns controles de iteração mais relevantes que serão descritos abaixo são o *while* e o *for*.

while

O controle de iteração *while* possui a seguinte lógica de repetição: “enquanto determinada condição for satisfeita, execute determinado código”. E sua sintaxe é a seguinte:

```
variavel = 1

while variavel <= 10:
```

```
#executa o trecho a ser repetido e, ao final e
#dentro da repetição, incrementa a variável
#contadora

variavel = variavel + 1
```

for

O comando *for* nos permite iterar sobre uma sequência de valores, como as listas e tuplas, e realizar operações sobre cada elemento desta sequência. Sua sintaxe é a seguinte:

```
#criação de uma lista

lista1 = [1, 2, 3, 4, 5]

# um a um, cada elemento da lista1 é armazenado em i e
#impresso na tela. Quando não houver mais elementos na
#lista, o comando for é encerrado

for i in lista1:

    print i
```

Criando e executando um programa em *python*

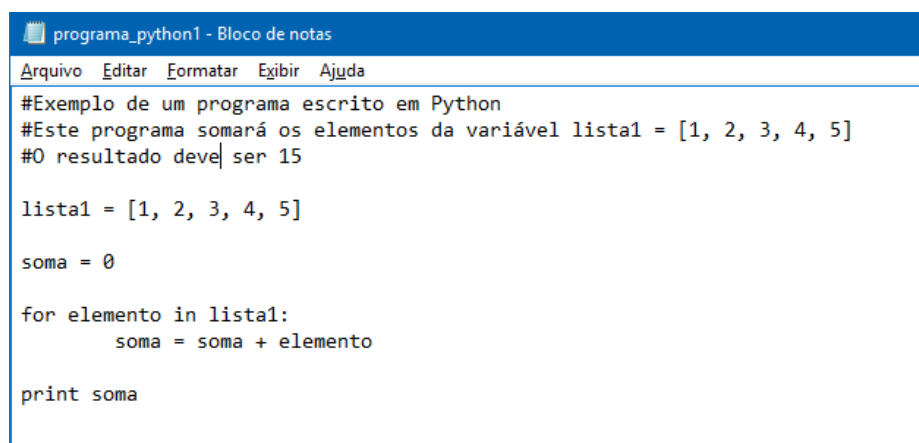
Neste tópico será explicado e exemplificado como se cria e executa um programa simples em python.

Para desenvolver um programa em python por mais simples que seja é necessário, no mínimo, um editor de texto puro como o *NotePad*, *TextPad*, *SublimeText*, etc. No *Linux* podem ser utilizados os editores de texto *Gedit* e *Xed*. Atenção aos processadores de texto como *Microsoft Word*, *WordPad*,

OpenOffice Writer, etc pois não servem como ambiente de desenvolvimento de software.

Escreva a lógica do programa no editor de textos seguindo as regras de desenvolvimento da linguagem python e salve o arquivo em um diretório qualquer com a extensão “.py”, por exemplo programa_1.py. É essencial que esta seja a extensão do arquivo, do contrário o interpretador python não reconhecerá o programa a ser executado.

Abaixo, segue um exemplo de um código em *python* que somará todos os elementos de uma lista de dados e apresentará na tela o resultado da soma. O arquivo foi salvo na Área de Trabalho do *Windows* e na Área de Trabalho do *Linux* com o nome programa_python1.py.



```
programa_python1 - Bloco de notas
Arquivo  Editar  Formatar  Exibir  Ajuda

#Exemplo de um programa escrito em Python
#Este programa somará os elementos da variável lista1 = [1, 2, 3, 4, 5]
#O resultado deve ser 15

lista1 = [1, 2, 3, 4, 5]

soma = 0

for elemento in lista1:
    soma = soma + elemento

print soma
```

Figura 23: Exemplo de Programa escrito em Python

Usaremos esse código para exemplificar como que se executa um arquivo *python* tanto no *Windows* quanto no *Linux*.

Executando um programa *python* no *Windows*

Usando o exemplo do arquivo acima, deveremos iniciar o *prompt* de comando, navegar até o diretório da Área de Trabalho e executar o arquivo `programa_python1.py`. Os passos em detalhes serão descritos abaixo:

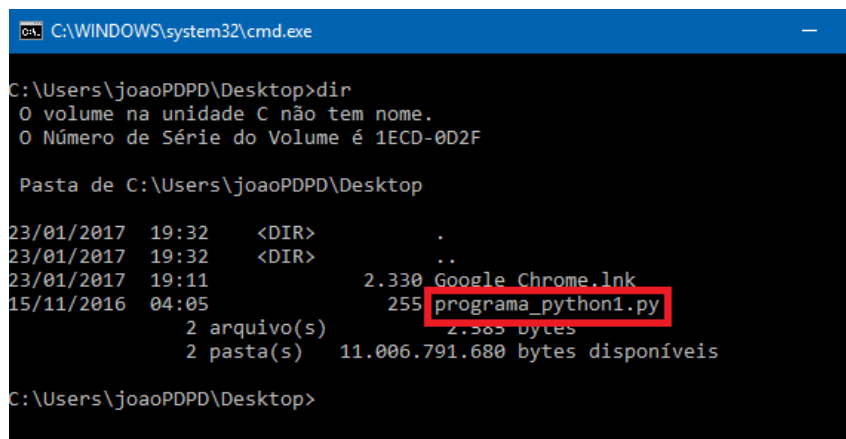
Abra o prompt de comando e utilize o comando `cd Desktop` para entrar no diretório da Área de Trabalho:



```
C:\WINDOWS\system32\cmd.exe
C:\Users\joaoPDPD>cd Desktop
C:\Users\joaoPDPD\Desktop>
```

Figura 24: Acesse a Área de Trabalho pelo prompt de comando

Uma vez no diretório Área de Trabalho, use o comando `Dir` para listar todos os arquivos e diretórios contidos na Área de Trabalho. Procure o arquivo de interesse nessa lista:



```
C:\WINDOWS\system32\cmd.exe
C:\Users\joaoPDPD\Desktop>dir
O volume na unidade C não tem nome.
O Número de Série do Volume é 1ECD-0D2F

Pasta de C:\Users\joaoPDPD\Desktop

23/01/2017  19:32    <DIR>          .
23/01/2017  19:32    <DIR>          ..
23/01/2017  19:11             2.330 Google.Chrome.lnk
15/11/2016  04:05             255 programa_python1.py
                2 arquivo(s)  2.585 bytes
                2 pasta(s) 11.006.791.680 bytes disponíveis

C:\Users\joaoPDPD\Desktop>
```

Figura 25: Verifique se o arquivo de interesse está no diretório

Agora, com a certeza de que o arquivo se encontra nesse diretório, execute o arquivo com o comando `python` seguido do nome do arquivo a ser executado: `python programa_python1.py`, conforme imagem abaixo:

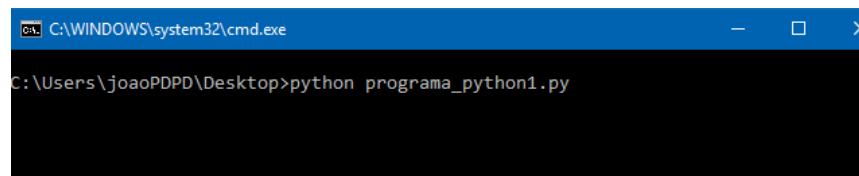


Figura 26: Execute o programa escrito em python

Depois de executado, ele apresentará o resultado da soma dos elementos da lista, ou seja, o valor da soma é 15 e ele apresentará esse valor na tela. Veja abaixo:

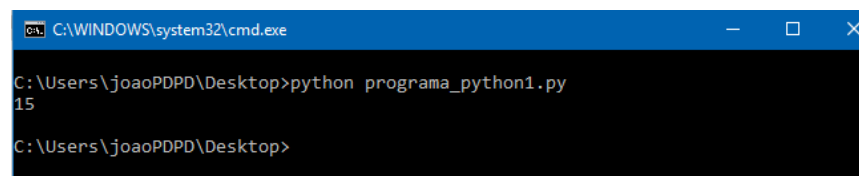


Figura 27: Resultado da execução do programa

Dessa forma, executa-se qualquer programa escrito em linguagem *python* no *Windows*.

Executando um programa *python* no *Linux*

Usando o exemplo do arquivo acima, deveremos iniciar o terminal do *Linux*, navegar até o diretório da Área de Trabalho e executar o arquivo `programa_python1.py`. Os passos em detalhes serão descritos abaixo:

Abra o terminal de comandos e utilize o comando `cd Área\ de\ Trabalho/` para entrar no diretório da Área de Trabalho:

```
/bin/bash
/bin/bash 80x24
joao@JoaoLinuxMint ~ $ cd Área de Trabalho/
joao@JoaoLinuxMint ~/Área de Trabalho $
```

Figura 28: Acesse a Área de Trabalho pelo prompt de comando

Uma vez no diretório Área de Trabalho, use o comando `ls` para listar todos os arquivos e diretórios contidos na Área de Trabalho. Procure o arquivo de interesse nessa lista:

```
/bin/bash
/bin/bash 80x24
joao@JoaoLinuxMint ~/Área de Trabalho $ ls
gcc datas.txt programa_python1.py
joao@JoaoLinuxMint ~/Área de Trabalho $
```

Figura 29: Verifique se o arquivo de interesse está no diretório

Agora, com a certeza de que o arquivo se encontra nesse diretório, execute o arquivo com o comando `python` seguido do nome do arquivo a ser executado: `python programa_python1.py`, conforme imagem abaixo:

```
/bin/bash
/bin/bash 80x24
joao@JoaoLinuxMint ~/Área de Trabalho $ python programa_python1.py
15
joao@JoaoLinuxMint ~/Área de Trabalho $
```

Figura 30: Resultado da execução do programa

Depois de executado, ele apresentará o resultado da soma dos elementos da lista, ou seja, o valor da soma é 15 e ele apresentará esse valor na tela, como na imagem acima.

Dessa forma, executa-se qualquer programa escrito em linguagem *python* no *Linux*.

Entrada e saída de dados (I/O – *input/output*)

Toda a seção de entrada e saída de dados a ser elucidada a seguir é baseada na documentação oficial do *Python, Library Reference* (2017) e *Language Reference* (2017), e na página *Standar Input, Standard Output and Standard Error* do *Linux Information Project*, LINFO (2006).

Os programas computacionais se comunicam com outros programas, com usuários ou com dispositivos de hardware através dos fluxos de entrada e saída de dados. Por exemplo, se um programa é desenvolvido para realizar a soma entre dois números quaisquer então é necessário que o agente executor do programa forneça como entrada dois números para, após a execução, receber como saída o resultado da soma.

Há formas diversificadas de se fornecer dados a um programa. Na linguagem *python*, assim como nas demais, é possível entrar com dados antes ou durante a execução de um programa, bem como a saída pode ocorrer enquanto o programa é executado ou quando é encerrado.

A seguir serão explicadas algumas formas de entrada e saída de dados mais significativas para a proposta deste trabalho.

Entrada padrão de dados

Convencionou-se dizer que a entrada padrão de dados (*standard input* ou *stdin*) é via teclado, ou seja, quando o usuário digita os dados para o programa. No python, há duas formas nativas de se fornecer dados a um programa que merecem ser destacadas: a entrada de dados por argumento e a entrada de dados pela função de *raw_input*.

Entrada de dados por argumento

Quando um programa é executado é possível, durante a sua invocação, passar dados no formato *string* como argumentos, ou seja, os dados são fornecidos ao programa antes mesmo da sua execução.

Como visto anteriormente, a execução de um programa python se dá pelo comando *python* seguido do nome do programa a ser executado. Esse é o método padrão de se invocar um programa quando não há dados previamente fornecidos. Do contrário, se houver a necessidade de iniciar o programa com dados disponibilizados antecipadamente, então a chamada ao programa se dará pela seguinte estrutura: *python <nome do programa> <argumento1> <argumento 2> ... <argumento n>*.

Uma vez passados os dados como argumentos na chamada do programa, é necessário que este seja capaz de resgatá-los para poder manipulá-los. Para tanto, o script deverá conter uma linha de código que, ao ser interpretada pelo python, permitirá que o programa utilize determinadas ferramentas de manipulação para esses dados. Dessa forma, a linha de código a ser inserida é a seguinte:

```
import sys
```

A função dessa linha de instrução é comunicar ao interpretador que o programa poderá fazer uso de determinadas variáveis e funções contidas no pacote de utilidades `sys`. Para esta situação, o pacote `sys` permite o uso da estrutura `sys.argv`, que é uma estrutura do tipo lista contendo todos os argumentos passados na inicialização do programa. Sendo `sys.argv` uma lista, então seus elementos podem ser acessados por meio da instrução `sys.argv[índice]`, em que *índice* é um valor inteiro maior ou igual a zero que corresponde à posição desses elementos na lista.

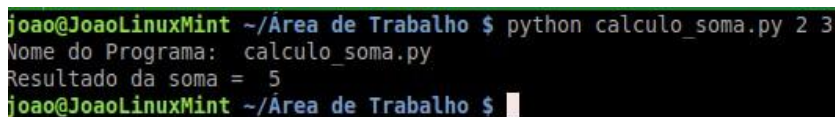
Por padrão, o conteúdo da lista na posição zero será sempre o próprio nome do programa. Os demais argumentos, quando passados, ocuparão a partir da posição 1 na lista `sys.argv`.

Por exemplo, suponha a existência de um programa chamado *calculo_soma.py*, cujo script é o seguinte:

```
1. import sys
2. nome_programa = sys.argv[0]
3. num1 = int(sys.argv[1])
4. num2 = int(sys.argv[2])
5. resultado = num1 + num2
6. print "Nome do programa: ", nome_programa
7. print "Resultado da soma = ", resultado
```

O código acima não solicita dados de forma explícita, como veremos no próximo tópico. Ao contrário, é esperado que o usuário forneça os dados na

chamada do programa para que ele consiga realizar a soma de dois números. A imagem a seguir mostra a chamada desse programa com os argumentos 2 e 3, resultando numa soma igual a 5.



```
joao@JoaoLinuxMint ~/Área de Trabalho $ python calculo_soma.py 2 3
Nome do Programa: calculo_soma.py
Resultado da soma = 5
joao@JoaoLinuxMint ~/Área de Trabalho $
```

Figura 31: Entrada de dados via argumento

Vale destacar na linha 2 que o argumento na posição zero é o próprio nome do programa. Nas linhas 3 e 4, as variáveis *num1* e *num2* receberão os argumentos opcionais passados na chamada do programa. Como mencionado anteriormente, esses argumentos são do tipo *string* e, portanto, é necessário que sejam convertidos para dados do tipo inteiro a fim de se realizar a operação de soma entre eles. Por fim, nas linhas 6 e 7, serão impressos na tela algumas informações. A função *print* utilizada nessas linhas será explicada detalhadamente no tópico sobre saída padrão.

Entrada de dados pela função de *raw_input*

Uma outra forma de entrar com dados em um programa é utilizando a função *raw_input([texto])*. Essa função é utilizada dentro do código fonte e é útil para fazer com que o usuário forneça dados ao programa enquanto este ainda está em execução. Quando o interpretador do *Python* encontra essa instrução no código, a execução do programa é parada (porém sem ocorrer o encerramento do programa) para que o usuário entre com algum dado. Assim que o dado é digitado e a tecla *enter* é pressionada o programa, então, retoma sua execução.

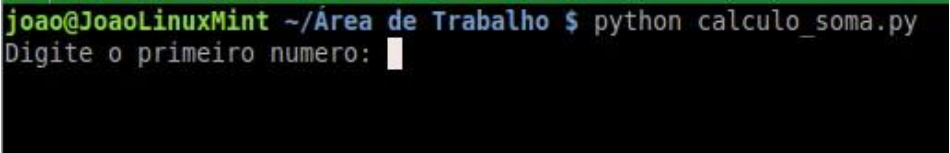
Nessa instrução `raw_input([texto])`, o conteúdo de `[texto]` é uma *string* e é facultativa. Se a *string* `[texto]` possuir algum conteúdo, então ele será impresso na tela e o usuário fará a digitação do dado após esse conteúdo impresso. Do contrário, se `[texto]` for uma *string* vazia, então não haverá nada a ser impresso mas o usuário poderá, ainda assim, entrar com algum dado.

Como essa função `raw_input([texto])` aguarda a entrada de um dado, é natural que se faça o uso dela combinado a uma variável, para que esse dado possa ser utilizado posteriormente.

A seguir segue o mesmo exemplo do código anterior, `calculo_soma.py`, porém utilizando a entrada de dados `raw_input([texto])`, ao invés da entrada por argumento.

```
1. num1 = int(raw_input("Digite o primeiro numero: "))
2. num2 = int(raw_input("Digite o segundo numero: "))
3. resultado = num1 + num2
4. print "Resultado da soma = ", resultado
```

Abaixo, segue as imagens da execução deste programa. Perceba que até que o usuário entre com o primeiro número e, o mesmo ocorre com o segundo número, a execução do programa é congelada e depois retomada.



```
joao@JoaoLinuxMint ~/Área de Trabalho $ python calculo_soma.py
Digite o primeiro numero: █
```

Figura 32: Programa aguardando entrada de dado


```
joao@JoaoLinuxMint ~/Área de Trabalho $ python calculo_soma.py
Digite o primeiro numero: 2
Digite o segundo numero: █
```

Figura 33: Programa continua e para aguarda nova entrada de dado

```
joao@JoaoLinuxMint ~/Área de Trabalho $ python calculo_soma.py
Digite o primeiro numero: 2
Digite o segundo numero: 3
Resultado da soma = 5
joao@JoaoLinuxMint ~/Área de Trabalho $ █
```

Figura 34: Programa termina execução e exibe o resultado

Saída padrão

Assim como há a entrada padrão de dados há, também, a saída padrão (*standard output* ou *stdout*), que no caso é a impressão visual dos dados na tela do usuário. No *python*, um modo de escrever dados na tela do usuário é utilizando a função *print*.

Saída de dados pela função print

A instrução *print* imprime dados em formato *string* na saída padrão, ou seja, ela apenas escreve caracteres na tela do usuário. Pode haver a situação em que o usuário queira escrever o conteúdo de uma variável que não seja do tipo *string* como, por exemplo, quando a variável for numérica. Neste caso, a

instrução *print* implicitamente realizará a conversão desse conteúdo para o tipo *string* para, em seguida, escrever o resultado na tela. A instrução *print*, para ser utilizada em um *script*, possui a seguinte estrutura: *print <string>*.

Há diversas maneiras práticas para a utilização dessa instrução, por exemplo quando *<string>* for um texto invariável, ou seja, quando uma *string* for passada de forma explícita para ser impressa na tela do usuário, como no exemplo a seguir:

```
print "Joao e aluno da UFABC"
```

```
>>> Joao e aluno da UFABC
```

Há também a situação em que se deseja imprimir apenas o conteúdo de uma variável na tela. Por exemplo, as variáveis *num1 = 10* e *nome = "Pedro Henrique"*. Nestes caso, a instrução *print* verificará se os conteúdos são do tipo *string*, converterá para *string* quando não forem e, por fim, imprimirá na tela do usuário o conteúdo das variáveis passadas. Segue abaixo os exemplos com uso de variáveis:

```
print num1
```

```
>>> 10
```

```
print nome
```

```
>>> Pedro Henrique
```

Por fim, há os casos em que se faz necessária a combinação entre *strings* explícitas com o uso de variáveis na instrução. Utilizando as variáveis

num1 e *nome*, empregadas no exemplo anterior, suponha que se deseja imprimir na tela do usuário a seguinte mensagem: “Pedro Henrique tem apenas 10 anos”. Isto pode ser feito combinando texto explícito com o uso das variáveis, como é demonstrado abaixo:

```
print nome, "tem apenas", idade, "anos"

>>> Pedro tem apenas 10 anos
```

Entrada e saída de dados via arquivos

Além da entrada e saída padrão de dados (via teclado), há outras maneiras de fornecer e obter dados de um programa, como por exemplo fazendo uso de arquivos. Um arquivo pode conter dados de natureza diversa como, por exemplo, arquivo de áudio, imagem, texto, etc. Devido ao escopo do presente trabalho serão explicados, a seguir, procedimentos referentes à manipulação de arquivos de texto.

Para este cenário há a possibilidade de se trabalhar tanto com arquivos em alto nível, ou seja, manipulando o texto em si (caractere por caractere), quanto em baixo nível, quando a manipulação é feita sobre os bytes que compõem o arquivo. Para cada caso, o nível de complexidade é diferente sendo mais simples e intuitivo a compreensão quando se trabalha em alto nível, que será o caso da explicação em seguida.

Entrada e saída de dados via arquivo de texto (leitura e escrita de arquivos)

A entrada de dados via arquivo requer que haja um arquivo cujo tipo de conteúdo seja textual, ainda que vazio. Ao invés de os dados serem digitados

manualmente, serão lidos pelo programa de forma automática. É importante indicar ao programa o local exato aonde se encontra o arquivo. Caso o arquivo de dados e o programa leitor estejam localizados no mesmo diretório, é suficiente indicar apenas o nome do arquivo para o programa.

Na maioria das linguagens de programação a manipulação de arquivos segue o seguinte protocolo: é preciso primeiramente localizar o arquivo para, em seguida, abri-lo. Uma vez aberto, a leitura poderá ser realizada e, ao final, quando não for mais necessária a manipulação do arquivo, deverá ser fechado.

Especificamente no *python* todo esse processo se resume à seguinte estrutura de código:

```
1. arquivo_ler = open("<caminho\nome do arquivo>",  
    "<modo de abertura>")  
2. #bloco de código realizando a leitura e manipulação  
3. #dos dados obtidos  
4. arquivo_ler.close()
```

Na linha 1 está descrito o processo de localização e abertura do arquivo. A função *open("<caminho\nome do arquivo>", "<modo de abertura>")* realiza a localização do arquivo através do parâmetro *<caminho\nome do arquivo>* que foi fornecido a ela. Uma vez localizado o arquivo, ela o abrirá de acordo com o parâmetro *<modo de abertura>*. Os dois parâmetros dessa função são do tipo *string*, logo devem ser passados entre aspas.

O parâmetro modo de abertura indica à função *open* o tipo de acesso que ela terá ao arquivo aberto, podendo ser:

“r”: *read mode* – modo leitura

Abre o arquivo para realizar somente leitura e posiciona o cursor no início do arquivo. Este é o modo de abertura padrão.

“r+”: *read/write mode* – modo leitura/escrita

Abre o arquivo e posiciona o cursor no início do arquivo para leitura ou escrita. A escrita nesse modo sobrescreve, no local do cursor, *substring* do texto original pela *string* que será escrita. Se o arquivo passado não existir no diretório, ocorrerá um erro de arquivo inexistente.

“w”: *write mode* – modo escrita

Abre o arquivo para realizar apenas escrita e posiciona o cursor sempre no início do arquivo. Se o arquivo ainda não existir no diretório, então ele será criado. Se existir, ele será totalmente apagado para poder ser reescrito.

“w+”: *write/read mode* – modo escrita/leitura

Abre o arquivo e posiciona o cursor no início do arquivo para escrita e leitura. Se o arquivo ainda não existir no diretório, então ele será criado. Se existir, ele será totalmente apagado para poder ser reescrito.

“a”: *appending mode* – modo junção (concatenação)

Abre o arquivo e posiciona o cursor no final do arquivo para concatenação, apenas. Como o cursor é colocado ao final do arquivo, este modo escreve no arquivo mantendo o que já existia. Se não existir o arquivo, então ele o cria.

“a+”: *appending/read mode* – modo junção/leitura

Abre o arquivo e posiciona o cursor no final do arquivo para concatenação e leitura. Como o cursor é colocado ao final do arquivo, este modo escreve no arquivo mantendo o que já existia. Se não existir o arquivo, então ele o cria.

Ainda na linha 1, após o arquivo ser aberto em um determinado modo, a variável *arquivo_ler* irá armazenar uma referência para esse arquivo. Quando isso ocorrer, a variável será o meio pelo qual o arquivo será manipulado, como será visto no exemplo a seguir.

As linhas 2 e 3 sintetizam o bloco de código para realizar a leitura do arquivo utilizando funções específicas a serem explicadas logo a seguir. A forma como a lógica de leitura será construída dependerá da necessidade de cada programa. Abaixo será explicado algumas funções importantes que auxiliarão na leitura de um arquivo texto:

Função *read([n])*

Esta função é responsável por ler do arquivo *n bytes* de uma vez (sendo cada byte um caractere, na codificação ASCII). O argumento *n* pode receber valores maiores ou iguais a 1.

Sempre que a função conseguir ler pelo menos um caractere, ou seja, pelo menos 1 *byte*, ela retornará uma *string* composta por todos os caracteres lidos em sequência.

Sempre que ela retornar uma *string* vazia, ou seja, uma *string* cujo tamanho é zero, significa que a leitura do arquivo chegou ao final (*End of File*).

Função *seek(offset[, whence])*

Essa função posiciona o cursor há uma distância *offset* de um determinado ponto de referência *whence*. O ponto de referência pode ser 0, 1 ou 2. Quando for zero, ou quando for omitido, significa que a referência é o início do arquivo. Quando for 1 a referência é a posição atual do cursor e, quando for 2 a referência é o fim do arquivo.

O argumento *offset* pode receber valores inteiros positivos, inteiros negativos ou zero. Valores negativos deslocam o cursor para a esquerda do referencial. Valores positivos deslocam para a direita do referencial e, quando for zero, não desloca o cursor.

Função *tell()*

Essa função devolve a posição atual do cursor.

Por fim, a linha 4 do modelo de código sinaliza ao interpretador *python* que o uso do arquivo aberto está encerrado, fechando-o, portanto.

Além da entrada de dados, há também a saída deles através da escrita de dados em arquivos. O protocolo de escrita de arquivos segue o mesmo padrão do de abertura: localização, abertura, escrita e encerramento do arquivo.

No *python* esse processo se resume à seguinte estrutura de código, semelhante à estrutura de leitura:

```
1. arquivo_escrever      =      open("<caminho\<nome      do
      arquivo>", "<modo de abertura>")

2. #bloco de código realizando a escrita dos dados no
```

```
3. #arquivo
4. arquivo_escrever.close()
```

Toda a explicação anterior sobre o protocolo de leitura é válida aqui. Acrescenta-se, então, a função relativa à escrita de dados em um arquivo:

Função *write(str)*

Essa função escreve uma *string str* no arquivo desejado.

Tratamento de erros de execução (try/except)

Exceções são erros no programa detectados durante sua execução, isto é, determinada instrução (linha de programa) é sintaticamente correta, porém somente durante a sua execução o erro será observado. Um exemplo clássico para esta situação é a divisão de um número por outro. No exemplo abaixo, faremos a divisão do conteúdo da variável *a* pelo conteúdo da variável *b*:

```
a = 10

b = 2

a / b

>>> 2
```

Sintaticamente está tudo correto e, durante sua execução, o cálculo foi efetuado com sucesso, resultando no valor 2 conforme esperado. Porém, se mudarmos o conteúdo da variável *b* para o valor 0, teremos um código sintaticamente correto e com um problema sendo percebido somente no momento de sua execução, como veremos a seguir:

```
a = 10
```



```
b = 2
```

```
a / b
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
ZeroDivisionError: integer division or modulo by zero
```

Note que durante a tentativa de executar a divisão o interpretador encontrou uma divisão por zero, acarretando em um erro no momento da execução da instrução denotado por *ZeroDivisionError*. As exceções quando não são tratadas geram uma interrupção imediata do programa, sendo algo indesejável.

Para contornar esse tipo de situação, existe uma estrutura chamada *Try/Except* que, quando utilizada, tenta executar um bloco de código e, caso ocorra uma exceção, um tratamento a esta exceção é realizado sem que a execução do programa seja interrompida.

Para a situação da divisão por zero, acima, a estrutura *try/except* poderia ser construída como no exemplo a seguir:

```
a = 10
```

```
b = 0
```

```
try:
```

```
    print a / b
```

```
except:
```

```
    print "Esta e uma divisao por zero"
```

O código acima encapsula dentro da instrução *try* a divisão de a por b . Sempre que b for diferente de zero, a divisão irá ocorrer normalmente e o resultado será impresso na tela (código dentro do bloco *try*). Porém, quando b for zero, o interpretador *python* não conseguirá realizar a divisão e a exceção ocorrerá, chamando assim o bloco *except*. No caso, o bloco *except* irá imprimir na tela a mensagem “Esta é uma divisão por zero” e o restante do programa continuará sendo executado, ou seja, não haverá interrupção indesejada do programa.

Redirecionamento

Foi visto até este ponto que o teclado e a tela do computador são, respectivamente, a entrada e a saída padrão de dados. Porém há meios alternativos de se fornecer e extrair dados de programas como, por exemplo, fazendo uso de arquivos.

Há, ainda, outra alternativa relacionada a esse fluxo de entrada e saída de dados que é o redirecionamento. Segundo LINFO (2005, *Redirection Definition*), o redirecionamento é a comutação de um fluxo de dados para que ele provenha de uma fonte diferente da fonte padrão ou para que ele seja direcionado para um destino distinto do padrão.

Referências

BARRETO, J. *Interpretadores, Compiladores e Tradutores*, 2001. Disponível em: <<http://www.inf.ufsc.br/~j.barreto/cca/arquitet/arg4.htm>>. Acessado em 15/11/2016.

COELHO, F.C *Computação Científica com Python: Uma introdução à programação para cientistas*. Edição do Autor, 2007.

LINFO. *Standard Input, Standard Output and Standard Error*, 2006. Disponível em: <<http://www.linfo.org/stdio.html>>. Acessado em 15/01/2017.

NIST. *Data Structure*, 2004. Disponível em: <<https://xlinux.nist.gov/dads/HTML/datastructur.html>>. Acessado em 15/11/2016.

OPEN SOURCE INITIATIVE. *The Open Source Definition*, 2007. Disponível em: <<https://opensource.org/osd>>. Acessado em 15/11/2016.

PYTHON 2.7.13 DOCUMENTATION. *Language Reference*, 2017. Disponível em <<https://docs.python.org/2/reference/index.html>>. Acessado em 20/01/2017.

PYTHON 2.7.13 DOCUMENTATION. *Library Reference*, 2017. Disponível em <<https://docs.python.org/2/library/index.html>>. Acessado em 20/01/2017.