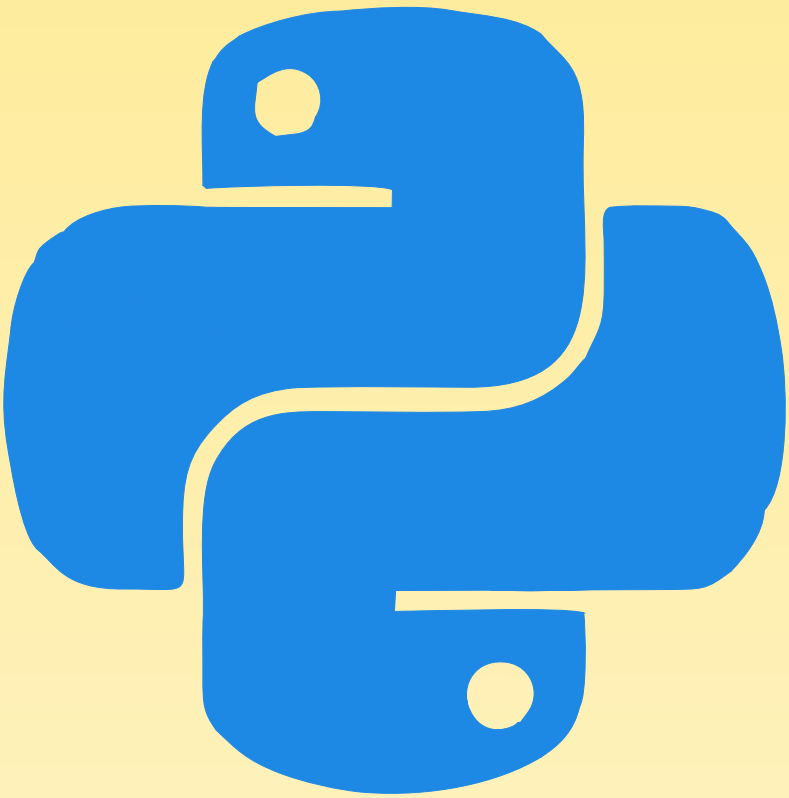
PETEE UFMG

UFMG

Learning.py

Uma apostila de introdução à programação em Python



2020

Programa de Educação Tutorial - Engenharia Elétrica
Universidade Federal de Minas Gerais

```
>>> title = 'MINICURSO DE PYTHON'
>>> group = 'Programa de Educação Tutorial da Engenharia
   Elétrica'
>>> ufmg = 'Universidade Federal de Minas Gerais'
>>> for i in (title, group, ufmg):
  print(i, end='\n\n')


MINICURSO DE PYTHON

Programa de Educação Tutorial da Engenharia Elétrica

Universidade Federal de Minas Gerais

>>> texto = open('texto1.txt', 'r')
>>> msg = texto.read()
>>> print(msg)
Esta apostila foi desenvolvida com o intuito de auxiliar
   no curso de Python oferecido pelo Programa de Educação
   Tutorial da Engenharia Elétrica(PETEE) da Universidade
   Federal de Minas Gerais(UFMG).

A apostila é material gratuito, atualmente disponível no
   site do PETEE.

A apostila introduz conceitos de programação utilizando a
   linguagem Python. Na primeira parte, são apresentados
   contextualizações e tutoriais de como instalar o
   Python. Na segunda, são abordados conceitos básicos de
   estruturas de dados, funções, operadores e controles de
   fluxo. Já na terceira parte, os conceitos da segunda
   são aprofundados, mostrando estruturas mais avançadas e
   mais noções de funções. Na quarta e última parte, há
   resoluções de exercícios e a bibliografia

Realizamos esta atividade com muito esmero.
>>> authors = 'Arthur Henrique Dias Nunes\nIsrael Filipe
   Silva Amaral'
>>> date = 'Janeiro de 2020'
>>> print(authors, date, sep='\n\n')
Arthur Henrique Dias Nunes
Israel Filipe Silva Amaral

Janeiro de 2020
```

PETEE UFMG

/peteeUFMG www petee.cpdee.ufmg.br @petee.ufmg

Grupo PETEE

O que é PET?

Os grupos PETs são organizados a partir de formações em nível de graduação nas Instituições de Ensino Superior do país. Estes grupos são orientados pelo princípio da indissociabilidade entre **ensino**, **pesquisa** e **extensão** e da educação tutorial.

Por esses três pilares, entende-se por:

- Ensino: as atividades extra-curriculares que compõem o Programa têm como objetivo garantir a formação global do aluno, procurando atender plenamente as necessidades do próprio curso de graduação e/ou ampliar e aprofundar os objetivos e os conteúdos programáticos que integram sua grade curricular.
- Pesquisa: as atividades de pesquisa desenvolvidas pelos petianos têm como objetivo garantir a formação não só teórica, mas também prática do aluno, de modo a oferecer as oportunidades de aprender novos conteúdos e de já se familiarizar com o ambiente de pesquisa científica.
- Extensão: vivenciar o processo ensino-aprendizagem além dos limites da sala de aula, com a possibilidade de articular a universidade às diversas organizações da sociedade, numa enriquecedora troca de conhecimentos e experiências.

PETEE UFMG

O Programa de Educação Tutorial da Engenharia Elétrica (PETEE) da Universidade Federal de Minas Gerais (UFMG) é um grupo composto por graduandos do curso de Engenharia Elétrica da UFMG e por um docente tutor.

Atualmente, o PETEE realiza atividades como oficinas de robôs seguidores de linha, minicursos de MATLAB, minicursos de LaTeX, Competição de Robôs Autônomos (CoRA), escrita de artigos científicos, iniciações científicas, etc.

Assim como outras atividades, o grupo acredita que os minicursos representam a união dos três

pilares: o pilar de ensino, porque ampliam e desenvolvem os conhecimentos dos petianos; o pilar da pesquisa, pois os petianos aprendem novos conteúdos e têm de pesquisar para isso; o pilar da extensão, porque o produto final do minicurso é levar à comunidade os conhecimentos adquiridos em forma de educação tutorial.

Minicurso de Python

A programação está muito presente não apenas no cotidiano dos petianos, mas de todos os graduandos em engenharia e também de diversos outros cursos superiores. Este também é um conhecimento que se torna cada vez mais presente com a disseminação de novos conteúdos em escolas, como ensino de robótica. Considerando agora o tema central do minicurso, a linguagem Python é considerada uma linguagem poderosa e fácil para se aprender, recomendada para o ensino de programação e atende vários níveis de programadores.

Tendo em vista este cenário, o Minicurso de Python foi desenvolvido para ser um curso de introdução à programação.

O grupo gostaria de agradecer, especialmente, a dois colegas e uma professora que contribuíram para o desenvolvimento da atividade, fornecendo materiais, dando sugestões, monitorias, indicações, dentre outros tipos de apoio. São eles: **Luiz Eduardo Borges Santana**, graduando de Engenharia Elétrica e monitor de Introdução a Programação de Computadores, ministrado pela professora **Camila Laranjeira** (Bibliografia 2); **Luiz Gustavo Almeida de Oliveira**, graduando de Engenharia de Controle e Automação.

As inscrições são abertas ao público e a apostila disponibilizada gratuitamente no site.

O Grupo

Tutora:
Luciana Pedrosa Salles
Discentes:
Álvaro Rodrigues Araújo
Amanda Andreatta Campolina Moraes
Arthur Henrique Dias Nunes
Diêgo Maradona Gonçalves Dos Santos
Gustavo Alves Dourado
Iago Conceição Gregorio
Isabela Braga da Silva
Israel Filipe Silva Amaral
Italo José Dias
José Vitor Costa Cruz
Lorran Pires Venetillo Dutra
Sarah Carine de Oliveira
Thais Ávila Morato
Tiago Menezes Bonfim
Vinícius Batista Fetter
Willian Braga da Silva

Contato

Site:
http://www.petee.cpdee.ufmg.br/

Facebook:
https://www.facebook.com/peteeUFMG/

Instagram:
https://www.instagram.com/petee.ufmg/

E-mail:
petee.ufmg@gmail.com

Localização:
Universidade Federal de Minas Gerais, Escola de Engenharia, Bloco 3, Sala 1050.

Agradecimentos

Agradecemos ao Ministério da Educação (MEC), através do Programa de Educação Tutorial (PET), Pró-Reitoria de Graduação da Universidade Federal de Minas Gerais (UFMG) e à Escola de Engenharia da UFMG pelo apoio financeiro e fomento desse projeto desenvolvido pelo grupo PET Engenharia Elétrica da UFMG (PETEE - UFMG).

Sumário

IV Índice

I Introdução

1. História

A linguagem Python foi desenvolvida em 1991 pelo holandês Guido van Rossum, Fig. 1.0.1. O nome Python foi devido ao costume do departamento onde foi criado de dar o nome em homenagem a algum programa de televisão. Sendo assim, o criador resolveu remeter à um de seus programas favoritos: *Monty Python's Flying Circus*, Fig. 1.0.2. Apenas depois a analogia com a cobra píton foi estabelecida e é usada no símbolo da linguagem, que são duas cobras entrelaçadas.

Figura 1.0.1: *Guido van Rossum em 2006*

Atualmente a propriedade intelectual da linguagem é mantida pela *Python Software Foundation*. `https://www.python.org/psf/`. Trata-se de uma corporação sem fins lucrativos com objetivos principais de promover, proteger e evoluir a linguagem.

Figura 1.0.2: *Monty Python's Flying Circus*

Python é uma **linguagem de programação**. As linguagens de programação podem ser entendidas como intermediárias ou tradutoras entre usuários(programadores) e computadores. Elas são artifícios usados para passar instruções para máquinas, isto é, programar. Essas instruções são passadas por meio de códigos, rotinas, *scripts* ou outros arquivos. Elas também exigem o cumprimento de regras semânticas, bem como pontuações adequadas, palavras chaves e até mesmo identação(principalmente em Python), essas regras são conhecidas como **sintaxe**, que varia dependendo da linguagem.

A linguagem Python é classificada como linguagem de propósito geral, que possui uma sintaxe elegante e tipagem dinâmica. Pode ser facilmente transformada para uma aplicação por meio da importação de bibliotecas. Também é uma linguagem **orientada a objetos**, ou seja, de **alto nível**. A hierarquia dos níveis de linguagens de programação pode ser observada na Fig. 1.0.3

As linguagens de alto nível podem ser classificadas em dois tipos: as interpretadas e as compiladas. O primeiro tipo necessita de um interpretador que irá transformar o código fonte em uma saída. Já o segundo, necessita um compilador para transformar o código fonte em arquivos de objeto, binários ou executáveis e de um executor para ler tais arquivos. Geralmente, os códigos compilados possuem melhor desempenho, em termos de tempo e processamento do que os interpretados.

O Python é uma linguagem interpretada, assim, não necessita de compilação. Nesta linguagem, a identação é fundamental, o código não irá funcionar se não estiver devidamente identado, pois é este artifício que indica quais estruturas estão subordinadas, diferente de C++ que utiliza chaves para isso.

Motivação

O Python é uma linguagem de programação de uso geral. Atualmente, é uma das linguagens mais populares no mundo, sendo usada nos mais diversos campos, tais como *machine learning*, inteligência artificial, desenvolvimento de jogos, pesquisas acadêmicas e ensino de programação.

É uma linguagem fácil de aprender, o que a faz uma boa opção para começar a programar. Uma das

Figura 1.0.3: *Níveis de Programação*

ideias fundamentais do Python é facilitar a criação de códigos que são simples de serem entendidos. Para isso, sua sintaxe é simples, clara e fácil de se entender.

Além disso, o Python é uma linguagem muito poderosa, sendo usada em diversos projetos e por várias grandes empresas, como por exemplo, Facebook, Google, NASA, Netflix, Dropbox e Instagram. Por causa de sua robustez e simplicidade, o Python é a linguagem de programação preferida por um grande número de *startups*.
Há um conjunto de princípios que influenciaram o desenvolvimento e design da linguagem, conhecidos por **Zen do Python**, análogo ao Zen do Budismo. Trata-se de dezenove orientações sobre a linguagem e é possível observá-lo pelo comando *import this*. Com base nele, é possível observar que a linguagem foi desenvolvida com ênfase em legibilidade e organização do código.

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and spanferably only one --obvious way to do
   it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Estes princípios são refletidos nas vantagens citadas anteriormente, como organização, robustez e simplicidade.

Sistemas operacionais como o Linux e o MacOS possuem o Python em seus terminais. Já no caso do Windows, é necessário instalar e adicionar ao *PATH*. Após a instalação, pode-se proceder como em qualquer outro sistema operacional: para acessar o Python, basta utilizar o comando *python* no terminal e para sair o comando *exit()*.

```
$ python
```

```
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC
   v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more
   information.
>>> print('Hello World')
Hello World
>>> exit()
$
```

Os procedimentos de como preparar o ambiente Python, instalações e atualizações serão abordados a seguir.

2. Preparação

2.1 Instalação

Como mencionado anteriormente, Linux e MacOS já possuem o Python em seus terminais. Assim, é necessário instalar apenas no Windows. Nos outros sistemas operacionais, talvez seja necessário apenas atualizar.

2.1.1 Windows

1. Para a instalação deve-se abrir o site oficial do Python: `https://www.python.org/` e ir em *Downloads*.

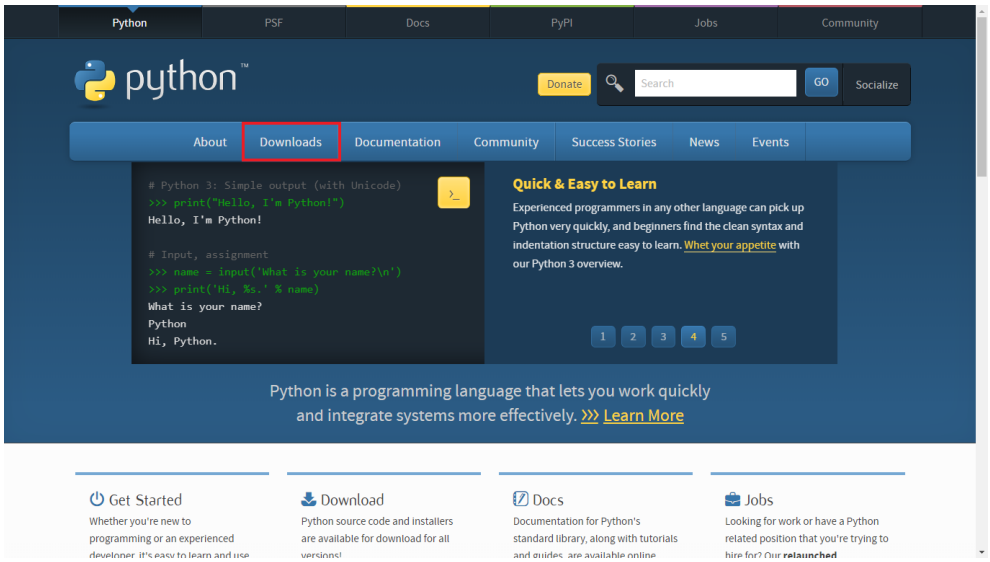


Figura 2.1.1: *Clique em Downloads*

2. Selecionar a versão e o instalador adequado para o sistema.

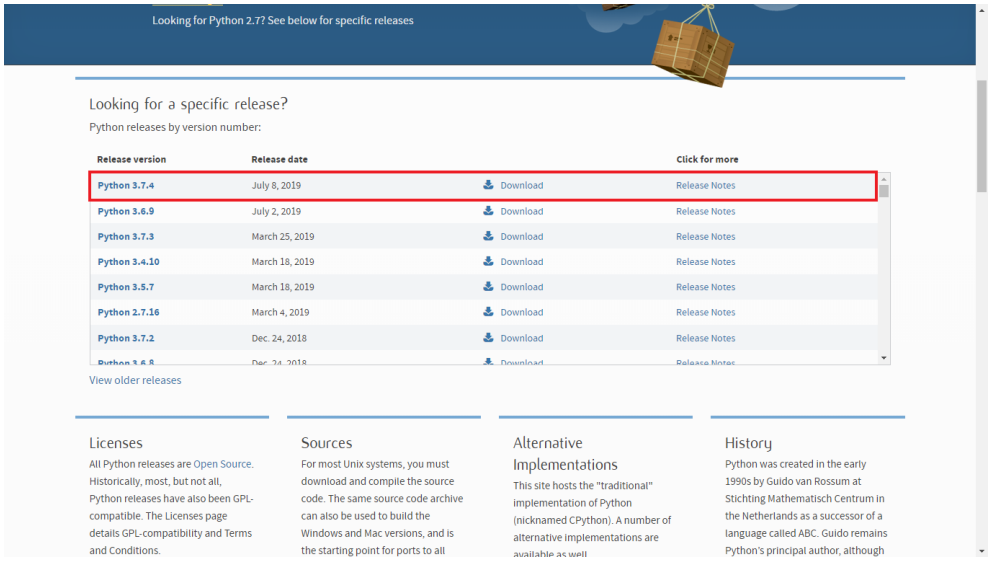


Figura 2.1.2: *É recomendado instalar a versão mais recente disponível*

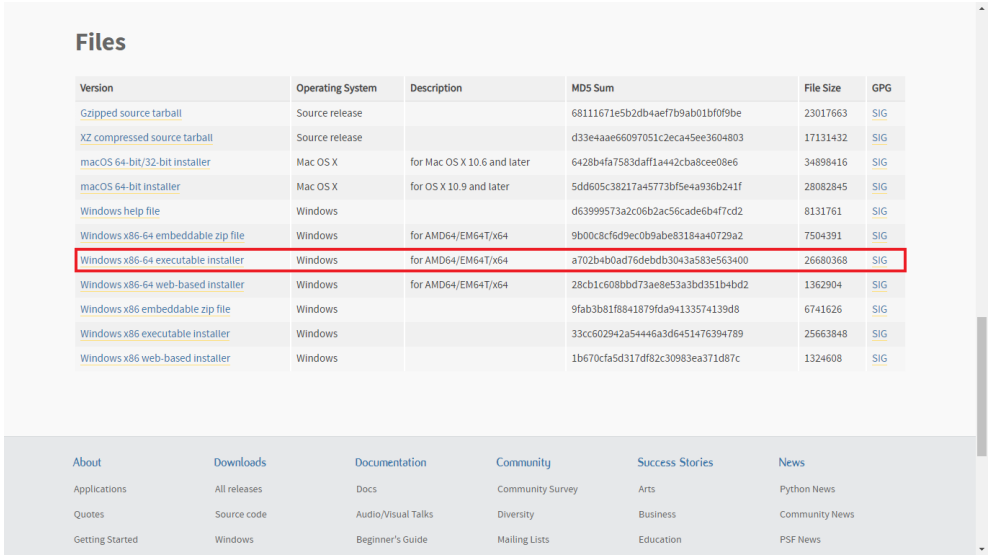


Figura 2.1.3: *Selecione a opção em destaque*

3. Por fim, basta abrir o instalador e clicar em *Install Now*. É recomendado marcar as duas caixas de seleção para que também seja possível acessar o Python pelos terminais do Windows.

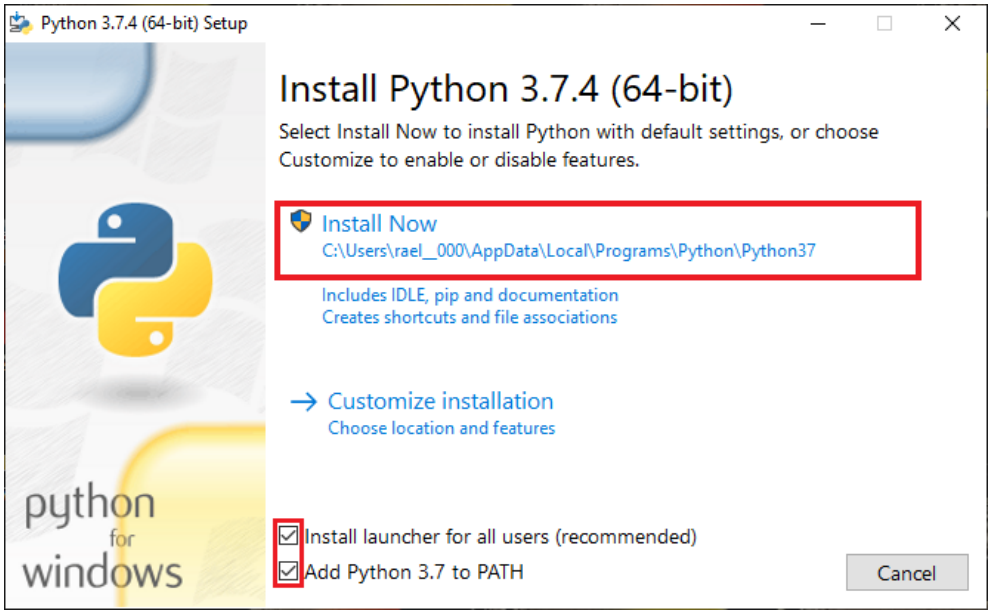


Figura 2.1.4: *Janela do instalador do Python*

Junto com o Python, também serão instalados alguns pacotes, o **IDLE** e o **pip**. O IDLE também é uma **IDE**, mais especificamente, a sigla é um acrônimo para *Integrated Development and Learning Enviroment*. A importância de IDEs será tratada a seguir, mas essa, em específico, foi criada em Python, podendo ser usada em Windows, Linux e MacOS. Já o pip é um pacote que serve para baixar e gerenciar outros pacotes que não estejam instalados.

2.2 Atualização

Eventualmente, há a necessidade de atualizar a versão da linguagem. Caso o usuário deseje, deve-se sempre ficar atento para não alterar a versão de seu próprio sistema. Para isso, deve-se proceder de formas diferentes dependendo do sistema operacional.

> **DICA:** A biblioteca *__future__* pode ser importada para usar algumas funcionalidades de versões mais atualizadas do Python. Importações serão tratadas em breve na página 38.

2.2.1 MacOS

Para atualizar a versão do Python no MacOS, deve-se acessar a aba *Downloads* no site oficial do Python: `https://www.python.org/`. Basta selecionar *MacOS*, a versão desejada e prosseguir com o instalador.

2.2.2 Linux

Já no Linux, a atualização se dá através do gerenciador de instalações. A distribuição de Linux mais comum é a Ubuntu. Neste caso, para atualizar deve-se utilizar o comando (Substituindo <X.X> pela versão desejada):

```
# Ex: sudo apt-get install python3.7
$  sudo apt-get install python<X.X>
```

É recomendado também instalar o gerenciador de pacotes do Python, o *pip*, via comando:

```
$  sudo apt-get install python-pip
```

Trocar a versão do Python do seu sistema operacional Linux irá comprometer algumas funções, como abrir o terminal ou instalar programas, é um erro comum. Por exemplo, trocar a versão do python 3 do 3.5 para o 3.6, principalmente quando se está aprendendo. Para resolver este erro pressione Ctrl + Alt + F7 para sair do modo GUI e entrar no modo CLI de seu sistema. Feito isso, selecione a versão recomendada (3.5) após digitar o comando

```
$ sudo update-alternatives -config python3
```

Para prevenir erros como este, é recomendável o uso de ambientes virtuais (VENVs) para a prática da linguagem.

2.3 IDEs

Para o aprendizado e desenvolvimento de softwares em linguagens de programação existem programas com funcionalidades e ferramentas integradas, que auxiliam os programadores. Esses programas são **Ambientes de Desenvolvimento Integrado**, do inglês Integrated Development Enviroment (IDE).

Para o desenvolvimento em Python, a IDE PyCharm, do desenvolvedor JetBrains, é bem conceituada e, por isso, recomendada. O download pode ser feito no site oficial da JetBrains, `https://www.jetbrains.com/pycharm/`. Há a versão paga *Professional*, com mais utilidades, e a versão gratuita para a comunidade estudantil e científica: *Community*.

A IDE ideal é aquela que atende aos gostos e necessidades do usuário. Outros ambientes mais amplos como *Atom* ou *Sublime Text* também são muito populares ou até mesmo um editor de texto para uma tela mais limpa como o *Notepad++*.

Os *scripts*, isto é, códigos podem ser executados dentro da IDE. Caso o usuário queira utilizar apenas um editor de texto, os scripts podem ser executados diretamente no terminal, como o exemplo a seguir:

O script: **meu_primeiro_programa.py**

```
1 print('Hello World')
```

No terminal:

```
# Certifique-se de que o arquivo
# esteja no diretorio corrente
python meu_primeiro_programa.py
```

> **IDE:** O próprio IDLE, instalado junto com o Python, é uma IDE.

2.4 VENVs

Além de prevenir erros no Python do seu sistema, o ambiente virtual pode ajudar a não encher o sistema de bibliotecas desnecessárias, bem como prevenir conflitos entre elas e suas versões durante o processo de aprendizado ou uso da linguagem. São mais usados em ambientes Linux.

O funcionamento de um ambiente virtual é simples, ele cria cópias de todos os diretórios necessários que o programa Python precisa, assim, as modificações serão feitas neste ambiente e não globalmente. Para a sua utilização é preciso instalar o pip e o virtualenv.

```
$ sudo apt install python-pip
$ sudo pip install virtualenv
```

Para criar um *virtualenv* e ativá-lo deve-se usar respectivamente os comandos:

```
$ virtualenv <nome>
$ source <nome>/bin/activate
```

Nota-se que o nome aparecerá como prefixo sempre que estiver dentro do VENV. Pode-se também desativá-lo ou removê-lo com os comandos:

```
$ deactivate
$ rm -r <nome>
```

> **IDE:** A IDE PyCharm possui seu próprio VENV, clicando em *Terminal* no canto inferior esquerdo.

2.5 Instalador de Pacotes - pip

O pip é o gerenciador de pacotes padrão do Python. Ele permite a instalação e o gerenciamento de pacotes adicionais que não fazem parte da biblioteca padrão. Basicamente, um pacote é um diretório que contém módulos, estes, por sua vez, são objetos que servem como uma unidade organizacional de código Python.

O pip é um programa que é utilizado via linha comando. Para usá-lo, a seguinte estrutura de comando é utilizada:

```
$  pip <argumentos>
```

Para instalar um pacote é possível usar um dos comandos a seguir:

```
$ pip install pacote              # Instala a ultima versao
$ pip install pacote==1.0.4       # Instala uma versao
   especifica
$ pip install 'pacote>=1.0.4'       # Instala a partir de
   uma versao minima
```

Para desinstalar um pacote basta digitar:

```
$  pip uninstall pacote
```

Caso deseje saber mais sobre todos comandos suportados pelo pip, digite:

```
$  pip help
```

Se não conseguir usar os comandos do pip diretamente é provável que o diretório atual não é onde foi instalado o *PATH* do sistema, então, basta usar o pip via o interpretador do Python:

```
python -m pip <pip arguments>
```

2.6 Shell vs Script

As linguagens de programação, descritas acima como tradutoras entre usuários e máquinas funcionam a base de comandos. Esses comandos são sequências de palavras reservadas e estruturas de códigos que seguem regras e traduzem o que o usuário quer para instruções de máquina. Por exemplo, para fazer com que a máquina imprima "Hello World!"na tela, devemos passar para a linguagem algo como "ImprimaNaTela ("Hello World!") ". As regras e sintaxe irão variar de linguagem para linguagem.

Quando as ações do programa se tornam mais complexas, o código responsável por

elas também se torna mais complexo. Uma característica importante das linguagens de programação é que os códigos, ou seja, as instruções que são passadas, são executados de forma **sequencial**. Ou seja, a linha 1 é executada primeiro, em seguida a 2 e assim por diante, seguindo um fluxo linear. Mais adiante, será abordado como fazer com que trechos desse fluxo sejam desviados ou repetidos, no capítulo de Controle de Fluxo.

A sequencialidade implica que ações como:

```
1 a = 2;
2 b = a + 1;
```

Irão ser executadas propriamente, enquanto que outras como:

```
1 b = a + 1;
2 a = 2;
```

São impossíveis de ser executadas, porque a primeira linha depende da execução da segunda.

Os conceitos de programação serão melhores destrinchados nos próximos capítulos. Por enquanto, tendo preparado o ambiente e entendido o que são linguagens de programação, o objetivo é familiarizar-se com a linguagem Python.

2.6.1 Shell

O termo Shell, sozinho, é popularmente usado para tradutores entre o sistema operacional e o usuário, conceito muito similar a própria definição de linguagens de programação. No caso do Shell, é comum que eles sejam linhas de comando, isto é, são prompts que executam os comandos usados pelo programador imediatamente, diferente dos códigos, que são organizados pelo programadores e depois executados. Exemplos de Shell são o *Bash* do Linux e a *Command Window* do Matlab.

Pode-se acessar o Shell do Python, chamado de **Python Interativo** ou de **Python Shell**, usando o comando *python* no terminal. Nota-se que , de fato, os comandos são executados imediatamente quando se pressiona a tecla *enter*. No caso do Python Shell, não é possível estruturar códigos e procedimentos mais complexos. Ele é melhor utilizado para testes de procedimentos simples e instruções rápidas. Para sair e voltar ao terminal, deve-se usar o comando *exit()*

```
$  python
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05)
   [MSC v.1916 64 bit (AMD64)] on win32
```

```
Type "help", "copyright", "credits" or "license" for
   more information.
>>>a=2
>>>b=3
>>>a+b
5
>>> print('Ola')
Ola
>>> exit()
$
```

Neste ponto, é importante frisar que, nesta apostila, quando se tratarem de códigos, o caractere cifrão "$"irá indicar que os procedimentos foram executados em um terminal. Pode ser o *Gnome-Terminal* do Linux, o *PowerShell* ou *CMD* do Windows, o terminal do MacOS, dentre outros. Já três caracteres *maior que* seguidos (>>>) indicarão que o código foi executado em Python Shell. Quando não houver marcadores serão códigos normais do Python, geralmente precedidos do nome do arquivo de código, com a extensão *.py*.

2.6.2 Script

Como visto até aqui, o Python Interativo não é capaz de armazenar sequências de comandos ou rotinas para serem executados posteriormente. Para isso, é preciso usar códigos, conhecidos como roteiros ou *scripts*, no caso do Python, chamado de **Python Script**.

Esses arquivos contém instruções para serem executadas. É importante afirmar novamente que os scripts são executados de formas sequenciais. Eles podem ser criados até mesmo pelo bloco de notas ou qualquer editor de texto e editados livremente. Para isso, basta salvá-los com a extensão *.py*.

meu_primeiro_programa.py

```
1 print('Hello World')
```

Para executá-los, basta abrir o terminal, navegar até onde estão salvos os arquivos desejados e usar o comando *python arquivo.py*:

```
# Certifique-se de que o arquivo
# esteja no diretorio corrente
$  python meu_primeiro_programa.py
```

IDE :Algumas IDEs possuem ferramentas como terminais internos que permitem executar o código dentro delas, sem precisar abrir terminais externos para testes.

2.7 Erros e Depuração

Há vários tipos de programadores, com hábitos, lógicas, aptidões e que trabalham com aplicações diferentes. Mas de uma maneira geral, um programador comete vários erros, isto é, dificilmente um programa complexo irá funcionar de primeira. O trabalho de um programador também envolve a habilidade de reler o código, identificar os erros e consertá-los, repetitivamente.

Os erros costumam ser de duas naturezas diferentes: erros de **sintaxe** e erros de **lógica**.

Os erros de sintaxe dizem respeito a digitação errada, ou identação incorreta ou marcações incorretas. Esses erros causam falhas na execução dos códigos e são, geralmente, indicados. É difícil prevení-los, quase impossível em programas muito longos, desta forma, o programador deve conferir as linhas de código em que foram apontados os erros, e como a linguagem é Python, deve-se caprichar na identação.

Já os erros de lógica acontecem quando o programa roda, mas não executa o que o programador gostaria que fizesse. Neste caso, deve-se revisar a lógica proposta e a lógica implementada. Em alguns casos, o erro pode ser de sintaxe, por exemplo, um procedimento que deveria estar subordinado a um loop não estar identado, assim, ele irá ser executado apenas uma vez e não repetidas.

Ademais, o programador deve desenvolver boas práticas de programação. A própria linguagem Python, pelos seus princípios, já favorece um código mais organizado e visualmente mais limpo, mas ainda sim, boas práticas devem ser desenvolvidas, como comentários no código e **modularização**.

DICA : Comentários são indicados pelo caractere cardinal "#", tudo o que estiver após ele em uma linha de código será ignorado pelo interpretador.

CURIOSIDADE: Na época de máquinas eletromecânicas, elas, eventualmente, tinham erro de execução porque haviam mariposas nos relês, daí originou-se o termo *bug*.

II Conceitos Básicos

3. Estrutura de Dados

Em algoritmos de computação é necessário armazenar informações diversas. Essas informações são traduzidas para dados de diferentes tipos e armazenados na memória. Cada bloco de memória que armazena um dado recebe uma etiqueta, um nome, e essas estruturas são chamadas de **variáveis** do programa. Isto é, nomes de controle que existirão no código para armazenar dados.

A este ramo da programação é dado o nome de estrutura de dados, que será introduzido neste capítulo. Estruturas compostas e afins serão vistos no capítulo Estrutura de Dados II.

3.1 Variáveis

Para o armazenamento, as informações devem ser comportadas em longas cadeias de dígitos binários, chamados de **bits**. Isto é, longas sequências de zeros e uns. Esses dados podem ser interpretados de diferentes tipos, dependendo do seu uso e das operações a serem realizadas com eles.

Seu armazenamento se dá em variáveis, que são espaços reservados na memória para fácil acesso do usuário. Todas as variáveis de Python são objetos, mas esse conceito será abordado futuramente, por hora, as classes serão tratadas como se fossem tipos.

CURIOSIDADE: O termo *bit* é uma sigla para se referir a dígito binário, do inglês, *B*inary dig*IT*.

3.1.1 Nomenclatura

A nomenclatura de uma variável, em Python, pode conter letras, números ou *underscore* "_", iniciando necessariamente com letras ou *underscore*. Como a maioria, a linguagem é **Case Sensitive**, isto é, diferencia caracteres minúsculos de maiúsculos.

Ainda assim, algumas palavras não podem ser usadas para nomear variáveis, pois são **palavras reservadas**, ou seja, palavras que indicam outros procedimentos para o interpretador. Algumas das palavras reservadas estão na tabela 3.1.1.

and	as	assert	break	class	continue
def	del	elif	else	except	exec
finally	for	from	global	if	import
in	is	lambda	nonlocal	not	or
pass	raise	return	try	while	with
yield	True	False	None		

Tabela 3.1.1: *Palavras Reservadas*

3.2 Tipos Primitivos

Os tipos de dados dizem respeito a forma com que as cadeias de 0s e 1s serão interpretados. Existem quatro tipos básicos comuns a maioria das linguagens, chamados tipos primitivos.

3.2.1 Inteiro

O primeiro tipo é o inteiro. Nesse tipo os dados são interpretados a fim de determinar o sinal e o valor, assumindo valores negativos, positivos e o zero, exatamente como no conjuntos dos números inteiros.

Em algumas linguagens há o tipo de inteiro *unsigned*, isto é, sem sinal, no qual só é possível armazenar valores não negativos. Com isso, o maior valor possível de se representar é o dobro+1 do tipo inteiro padrão.

3.2.2 Ponto Flutuante

O tipo ponto flutuante é responsável por representar o conjunto dos números reais. Pode-se também representar inteiros, mas não tantos quanto o tipo inteiro, pois a capacidade de armazenamento deste tipo está direcionada para poder suportar valores decimais.

Neste tipo, é possível representar valores como 2.5, -0.3333, etc.

3.2.3 Lógico

Já o tipo lógico armazena os valores **booleanos**, ou seja, verdadeiro ou falso, 0 ou 1.

3.2.4 Caractere

Por fim, o tipo caractere é responsável por armazenar valores de letras e outros caracteres.

Geralmente, os caracteres são codificados em valores seguindo um padrão universal, conhecido como Tabela ASCII.

3.3 Tipos do Python

Analogamente aos tipos primitivos, o Python possui as classes **int** para inteiros, **float** para pontos flutuantes, **bool** para booleanos e **str** para *strings*, isto é, sequências de caracteres. Não há um tipo simples para trabalhar com caracteres, como o tipo *char* de outras linguagens. No Python os caracteres são trabalhados por arranjos, em uma estrutura composta chamada de string. Adicionalmente, há a classe **complex** para representar números complexos.

Python é uma linguagem de **tipagem dinâmica**, isto é, dependendo do valor atribuído a uma variável, seu tipo é definido dinamicamente, podendo ser alterado durante o código. O comando *type(var)* retorna a classe de var.

```
>>> Frase = 'Esta e uma frase'
>>> type(Frase)
<class 'str'>
>>> a = 4
>>> type(a)
<class 'int'>
>>> x = 2.5
>>> type(x)
<class 'float'>
>>> Q = True
>>> type(Q)
<class 'bool'>
>>> s = 1 + 2j
>>> type(s)
<class 'complex'>
```

DICA: A presença do caractere ponto já indica que o usuário deseja que o valor seja armazenado em float, mesmo que o valor seja inteiro. Exemplo: se digitado
x = 4.
x será do tipo float.

DICA: O valor de uma variável pode ser impresso digitando apenas o nome dela. Exemplo:
x = 4
x
4

Os dados podem ser convertidos para outros tipos usando-se o nome do tipo desejado e o valor como argumento.

```
>>> c = str(19)
>>> c
'19'
>>> x = int(c)
>>> x
19
```

Mesmo os tipos sendo implicitamente definidos, o programador pode definí-los usando o nome do tipo e o valor como argumento, ou seja, fazendo uma conversão:

```
>>> x = float(3)
>>> a = str(x)
>>> x = x + 1
>>> a
'3.0'
>>> print('Valor = ' + a)
Valor = 3.0
>>> print('Valor = ', x)
Valor =  4.0
```

DICA: Em Python, as aspas duplas também podem ser usadas para definir strings, mas geralmente são usadas aspas simples como boas práticas de programação.

DICA: Note que é possível usar uma para escrever a outra como caractere: ' "' ou " "'.

DICA: Deve-se ter atenção para essa sintaxe, porque apesar de para o Python aspas simples ou duplas terem o mesmo efeito, em outras linguagens elas podem ser fundamentalmente diferentes. Em C, por exemplo, as duplas definem arranjos de caracteres, enquanto que as simples definem caracteres únicos. Em PHP, as duplas definem tipos dinâmicos e as simples tipos estáticos.

Valores numéricos são geralmente atribuídos na notação de base decimal, mas é possível atribuir valores na notação de binários, de base oito e de hexadecimais com os prefixos *0b*, *0o* e *0x*, respectivamente.

```
>>> d = 31 #base decimal
>>> d
31
>>> b = 0b11111 #base binaria
>>> b
31
>>> o = 0o37 #base octal
>>> o
31
>>> h = 0x1F #base hexadecimal
>>> h
31
```

DICA : Valores em bases específicas podem ser dados usando-se o comando int('valor', base = X), sendo que o valor precisa estar obrigatoriamente em string.
Exemplo:
>>> x = int('11', base=3)
>>> x 4

Como as estruturas de dados abordadas aqui são classes, elas possuem métodos. A **orientação a objetos** será abordada futuramente, mas por hora, os métodos serão tratados como funções e procedimentos inerentes a toda variável. Por exemplo, uma *string* que contém uma frase pode ser separada pelas suas palavras pelo método *split()*. O uso dos métodos se dá da seguinte forma: *Variável.método()*.

```
>>> Frase = 'Uma frase'
>>> Frase
'Uma frase'
>>> Frase.split()
['Uma', 'frase']
```

```
>>> a, b = Frase.split()
>>> a
'Uma'
>>> b
'frase'
```

Para visualizar todos os atributos e métodos da classe deve-se utilizar o comando *dir(classe)* ou *dir(variavel)*.

Python também possui outras classes, estruturas compostas. São elas **listas**, **tuplas**, e **dicionários**, além das **strings**. Essas estruturas serão tratadas futuramente na página 59.

Exercícios

3.1 Em um determinado programa são necessárias variáveis de diferentes tipos para armazenar diferentes informações. Em cada caso a seguir informe qual tipo de dado primitivo seria mais adequado para o armazenamento. Informe também qual o tipo do Python correspondente:
a) Idade de uma pessoa;
b) Se a pessoa é maior de idade ou não;
c) Altura de uma pessoa;
d) Peso de uma pessoa;
e) Primeira letra do nome de uma pessoa;
f) Última letra do nome de uma pessoa.

3.2 Considere as variáveis: a = 1; b = 2; c = ’1’; d = ’2’. Utilizando o operador "+"observe a diferença entre somar a + b e somar c + d. Qual a diferença e por que isso ocorre?

3.3 Ainda considerando as variáveis da questão anterior, o que poderia se fazer para que a + b tenha o mesmo resultado de c + d ou vice versa?

3.4 Em cada item a seguir diga se pode ser um nome de variável. Em caso negativo, informe o motivo:
a) x;
b) aux;
c) 1aluno;
d) aluno1;
e) aluno_1;
f) aluno 1;
g) _aluno1;
h) int;
i) Int;
j) int_idade;
k) class.

```
def getInspiration(morningDay):
    if morningDay == 'depressed':
        start.coding() and be.awesome()
```

4. Comandos e Funções

Entendido as estruturas de dados, é necessário compreender os comandos e funções para melhor utilizá-los. Este capítulo introduzirá os comandos e funções. Conteúdos mais complexos, bem como funções definidas pelo usuário e leitura/escrita de arquivos serão vistos no capítulo Comandos e Funções II.

Os comandos são palavras chaves que executam certas ações e as funções são procedimentos que devolvem dados dependendo de seus argumentos. Os conceitos são parecidos e as vezes confundidos. No Python, ainda, devido a orientação de objetos, alguns procedimentos, chamados de funções, são ainda métodos de objetos, mas essa distinção não será crucial para este capítulo.

As funções são similares a funções matemáticas, isto é, dependendo do argumento, o resultado é diferente. O uso de funções em algum momento no script ou no Python interativo recebe o nome de **chamada**. Quando uma função é chamada, geralmente recebe **argumentos**, que são passados como **parâmetros**, dentro de parênteses. As respostas da função, quando há, são chamados de **retornos**.

Desta forma temos que

$$retorno1, retorno2 = funcao(argumento1, argumento2)$$

Por exemplo, ao usar a função de raiz quadrada *sqrt* com argumento 25, é retornado o valor 5, ou seja, para:

```
>>> x = sqrt(25)
```

A variável x receberá o valor 5.

O próprio usuário pode criar funções para serem utilizadas no código. Esse processo é denominado **definição de funções**. A definição de funções será abordada mais a frente, porque, para isso, é necessário o conhecimento de **controle de fluxo**, isto é, estruturas condicionais, de repetição e aninhamento de estruturas.

4.1 Importação

A linguagem Python é conhecida pelas diversas áreas de aplicações. Por vezes, tais áreas necessitam de comandos ou funcionalidades específicas que não estão, por padrão, no Python, elas são advindas de **bibliotecas** e precisam ser importadas, caso o usuário deseje utilizá-las.

Para isso é utilizado o comando *import*.

Por exemplo, a função *sqrt* para cálculo de raízes quadradas está na biblioteca *math*. Portanto, deverá ser impresso um erro ao utilizá-la sem importar a biblioteca.

```
>>> sqrt(9)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'sqrt' is not defined
```

Para corrigir o erro, antes é necessário usar o comando *import math*.

Ainda assim, ao utilizar as funcionalidades advindas da biblioteca deve-se usar o nome da biblioteca. No caso:

```
>>> import math
>>> sqrt(9)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'sqrt' is not defined
>>> math.sqrt(9)
3.0
```

4.1.1 Importação Otimizada

O comando *import* importa a biblioteca com todas as suas funcionalidades. As vezes, apenas algumas funcionalidades são de interesse, como no exemplo acima, apenas a *sqrt*. Há a opção de importar apenas o que se deseja para economizar memória e para não

precisar especificar a biblioteca da qual foi importado. Para isso, em conjunto com o comando *import*, utiliza-se o comando *from*, de forma a ficar como: from <biblioteca> import <funcionalidade1>, <funcionalidade2>,...

Dessa forma, os métodos são importados diretamente para a pasta, por isso não é necessário explicitar a biblioteca de origem.

```
>>>from math import cos, pi
>>>cos(pi)
-1.0
```

DICA: Dentro do site oficial do Python, na aba PyPI, é possível acessar o index de pacotes do Python, isto é, todas as bibliotecas e projetos possíveis de se importar. `https://pypi.org/`. É possível criar uma própria biblioteca e disponibilizar na comunidade para livre acesso.

IDE: Ctrl + espaço na IDE PyCharm mostra todas as opções dentro da biblioteca.

IDE: Para gerenciar os pacotes e bibliotecas instalados pela IDE PyCharm, deve-se ir em *Settings* ou *Preferences*, selecionar o projeto e selecionar *Project Interpreter*. Lá, será possível visualizar as bibliotecas instaladas, adicionar ou remover bibliotecas.

4.1.2 Outras Bibliotecas

Algumas bibliotecas são **built-in**, isto é, já são instaladas por padrão junto com o Python, como a biblioteca math e random. A importação delas é feita naturalmente. No entanto, ao tentar importar bibliotecas que não são built-in, deve-se antes instalá-las na máquina. Algumas IDEs já fazem o gerenciamento e possuem atalhos para facilitar este processo, podendo importá-las também naturalmente.

Mas, se for o caso do usuário não usar IDEs, ou usar o IDLE, por exemplo, deve-se fazer o download do pacote. Para isso, é recomendado usar o pip. Para instalar uma nova biblioteca, navegue até o diretório padrão de instalação do Python, abra o terminal e digite o comando:

```
$  pip install <bib>
```

Assim, a nova biblioteca poderá ser usada. Para remover o pacote, deve-se usar o comando:

```
$  pip uninstall <bib>
```

Algumas bibliotecas serão citadas no capítulo 11 a título de curiosidade.

4.2 Entrada e Saída de Dados

Para se criar rotinas e procedimentos mais sofisticados, por vezes, é necessário programar para coletar dados do usuário e imprimir os resultados. Isto é chamado de entrada e saída de dados.

4.2.1 Entrada

A função canônica de entrada de dados, isto é, para pedir informações do usuário é a função *input*.

Nela, o argumento é uma mensagem impressa e o retorno é o que o usuário digitar.

```
>>> frase = input('digite uma frase\n')
digite uma frase
Ola
>>> frase
'Ola'
```

Nesta função, mesmo que o usuário digite valores numéricos, os dados serão armazenados como strings. Para usá-los como outros tipos de dados, deve convertê-los.

```
>>> x = int( input('digite um valor\n') )
digite um valor
2
>>> x
2
```

DICA: No Python 2, a função *input* não trata a expressão como string, apenas a avalia. Para utilizar como a função input das versões posteriores, deve-se utilizar a função *raw_input*.

4.2.2 Saída

Para a saída de dados há a função *print*. Pode imprimir mais de uma string e também valores, separados por vírgulas.

```
>>> print(1, 'mais', 1, 'igual a', 2)
1 mais 1 igual a 2
>>>
```

A função print possui dois parâmetros fixos, os quais a especificação não é obrigatória. O parâmetro separador e o parâmetro de fim. O separador será a string que separa os diferentes argumentos usados, que na chamada são separados por vírgula. Já o parâmetro de fim é a string impressa no fim da função print. Se não forem especificados, o separador será um espaço simples e o fim será quebra de linha "\n". Para especificar o separador deve-se usar o argumento *sep='string'* e para especificar o fim *end='string'*, em que 'string', deve ser substituída pela string que o usuário quiser usar.

```
>>> A = 'Uma frase'
>>> B = 'separada'
>>> x = 123
>>> print(A, B, x)
Uma frase separada 123
>>> print(A, B, x, sep='-')
Uma frase-separada-123
>>> print(A, B, x, end='nada')
Uma frase separada 123nada>>> print(A, B, x, sep='---',
   end='\nfim\n')
Uma frase---separada---123
fim
>>>
```

Para se imprimir variáveis, há varias formas de fazê-lo.

O exemplo irá supor duas variáveis, sendo elas nome e nota:

```
>>> nome = 'Joao'
>>> nota = 10
```

- Passar como uma **tupla**. Tuplas são estruturas de dados compostas que serão explicadas posteriormente. Neste modo, os locais na string que serão substituídos por variáveis são indicados com % seguidos de uma letra.

    ```
    >>> print('A nota de %s foi %s' % (nome, nota))
    A nota de Joao foi 10
    ```

- Passar como um **dicionário**. Dicionários também são estruturas compostas explicadas posteriormente. Este modo é parecido com o anterior, a diferença é que são postas etiquetas dentro de parênteses para indicar a variável.

```
>>> print('A nota de %(n1)s foi %(n2)s' % {'n1':
    nome, 'n2': nota})
A nota de Joao foi 10
```

DICA: Nos dois primeiros modos, é usado % ou %() seguido de algum caractere. O caractere indica a forma de impressão da variável, sendo:
%s para string;
%d ou %i para inteiros;
%f para valores decimais, podendo especificar que a impressão tenha X casas depois da vírgula usando %.Xf;
%e para notação científica %g para o mais resumido entre científica ou inteiro/decimal.

- Usar *.format*. Neste modo, onde seriam as variáveis deve-se substituir por chaves "{}"e explicitar as variáveis em seguida.

```
>>> print('A nota de {} foi {}'.format(nome, nota))
A nota de Joao foi 10
```

Nesse caso, também pode-se colocar etiquetas para especificar depois ou explicitar a ordem em que os elementos serão citados:

```
>>> print('A nota de {n1} foi
    {n2}'.format(n1=nome, n2=nota))
A nota de Joao foi 10
>>> print('A nota de {0} foi {1}'.format(nome,
    nota))
A nota de Joao foi 10
>>> print('A nota de {1} foi {0}'.format(nota,
    nome))
A nota de Joao foi 10
```

- Usar f-strings. Similar ao uso de format, porém mais compacto.

```
>>> print(f'A nota de {nome} foi {nota}')
A nota de Joao foi 10
```

OBS: O modo de usar f-strings só está disponível a partir da versão 3.6.

- Simplesmente passar as variáveis como parâmetros.

```
>>> print('A nota de', nome, 'foi', nota)
A nota de Joao foi 10
```

- Converter para strings e usar o operador "+"para concatená-las. Deve-se lembrar também dos espaços em branco. No exemplo, a variável nome já é uma string.

```
>>> print('A nota de ' + nome + ' foi ' +
    str(nota))
A nota de Joao foi 10
```

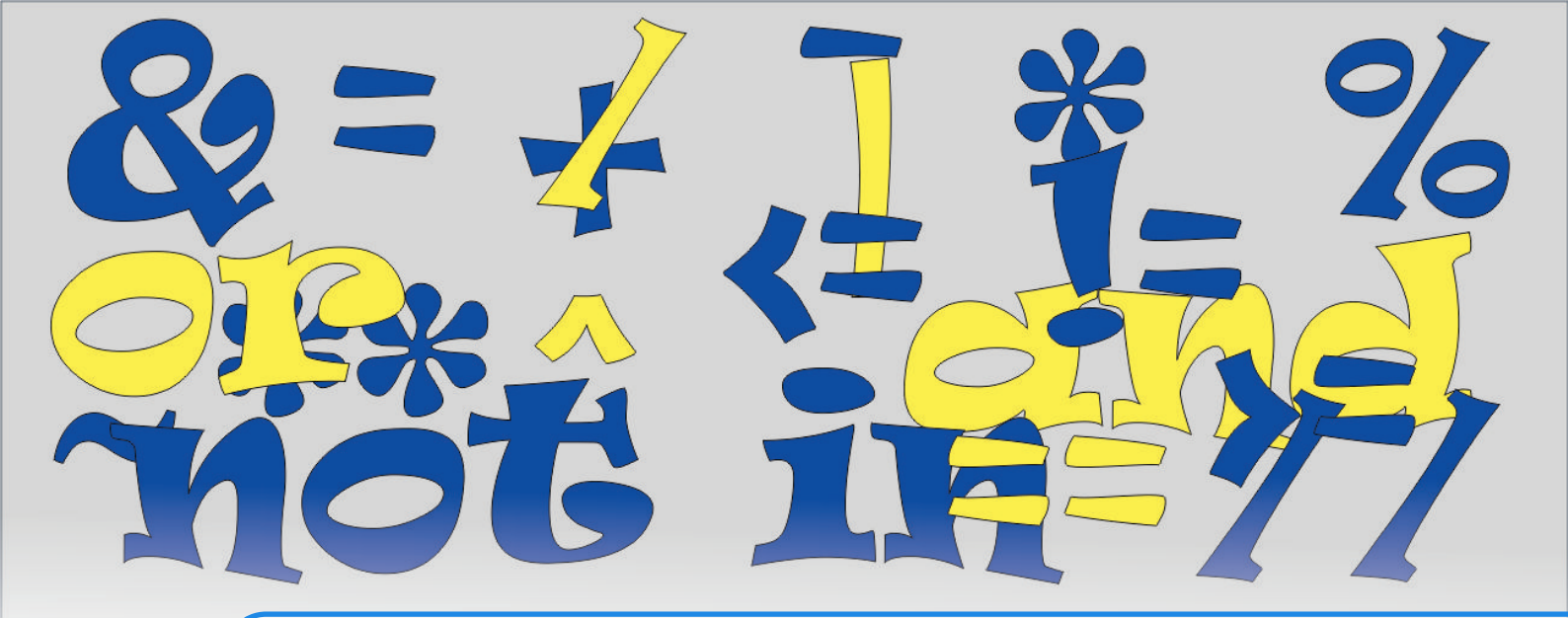
DICA: No Python 2 as opções de usar *.format*, f-strings e de passar as variáveis como parâmetros não estão disponíveis, porque print não é uma função nesta versão.
Para usá-la desta forma, pode-se utilizar o comando
from __future__ import print_function

DICA: Para evitar que as contrabarras sejam interpretadas como caracteres especiais, como \n e \t, pode-se usar o prefixo *r* antes da string ou utilizar contrabarras duplas \\
print(r'C:\temp\new')
ou
print('C:\\temp\\new')

Exercícios

4.1 Peça do usuário um valor em graus para um ângulo. Converta-o para radianos e, usando funções da biblioteca math, imprima o seno, cosseno e tangente deste ângulo.

4.2 Repita o processo da questão anterior, porém utilizando a função deg2rad da biblioteca numpy para converter o ângulo de graus para radianos.



5. Operadores

Este capítulo irá abordar os operadores do Python, isto é, os elementos do código responsáveis por indicar as ações a serem feitas com dados. Alguns já foram abordados anteriormente, de forma implícita.

Deve-se atentar que, como as variáveis são objetos, os operadores podem realizar operações diferentes, dependendo do tipo de dado usado. Por exemplo, o operador aritmético de adição realiza soma entre dois números, no entanto, entre strings realiza concatenação. Sendo assim, as descrições a seguir são para variáveis numéricas.

5.1 Aritméticos

Os operadores aritméticos são responsáveis por realizar as operações matemáticas básicas, eles estão representados na tabela a seguir:

Exemplo:

```
>>> a = 5; b = 3
>>> a + b
8
>>> a/b
1.6666666666666667
>>> a//b
1
```

Tabela 5.1.1: *Operadores Aritméticos*

Tipo	Operador	Descrição
Aritmético	+	Adição
Aritmético	-	Subtração
Aritmético	*	Multiplicação
Aritmético	/	Divisão
Aritmético	//	Divisão inteira
Aritmético	%	Resto da divisão
Aritmético	**	Potenciação

5.2 Lógicos

Os operadores lógicos são muito utilizados em estruturas condicionais e estruturas de repetição. Esses operadores avaliam expressões lógicas e retornam verdadeiro ou falso. Por falso, entende-se o valor nulo, zero, e por verdadeiro, qualquer valor diferente de zero, por padrão, o valor um.

Exemplo:

```
>>> 0 and 1
0
>>> not 0
True
>>> 2 and 2
2
```

Tabela 5.2.1: *Operadores*

Tipo	Operador	Descrição
Lógico	not	Não
Lógico	and	E
Lógico	or	Ou

5.3 Relacionais

Os operadores relacionais avaliam expressões e retornam verdadeiro ou falso, assim como os lógicos. No entanto, os relacionais são responsáveis por operações de comparação de magnitude.

Exemplo:

```
>>> 0 == 0
```

```
True
>>> 2 != 2
False
>>> 3 <= 5
True
```

Tabela 5.3.1: *Operadores*

Tipo	Operador	Descrição
Relacional	==	Igual a
Relacional	!=	Diferente de
Relacional	<>	Diferente de
Relacional	>	Maior que
Relacional	>=	Maior ou igual a
Relacional	<	Menor que
Relacional	<=	Menor ou igual a

5.4 Atribuição

Os operadores de atribuição realizam a ação de alocar valores para variáveis. Podem ser usados para realizar operações aritméticas e alocar o resultado simultaneamente.

Exemplo:

```
>>> x = 4; y = 2
>>> x += y #equivale a x = x + y
>>> x
6
```

Tabela 5.4.1: *Operadores*

Tipo	Operador	Descrição
Atribuição	=	Atribuição
Atribuição	+=	Atribuição adição
Atribuição	-=	Atribuição subtração
Atribuição	*=	Atribuição multiplicação
Atribuição	/=	Atribuição divisão
Atribuição	//=	Atribuição divisão inteira
Atribuição	%=	Atribuição resto
Atribuição	**=	Atribuição potenciação

5.5 Bit a bit

Para se compreender as operações binárias, deve-se antes, entender um pouco sobre a representação de dados e instruções em computadores. Para eles, os dados são representados por meio de cadeias de dígitos binários, os **bits**. Cada bit pode assumir o valor de 0 ou 1. Um número, por exemplo, na representação do cotidiano é na base decimal. Para o computador, esse número precisa ser convertido para base binária.

Por exemplo, supondo uma representação de números naturais usando quatro bits, os números ficariam:

- 0000 -> 0
- 0001 -> 1
- 0010 -> 2
- 0011 -> 3

- ...
- 1111 -> 15

Desta forma, os operadores bit a bit realizam operações com cada bit dos dados afim de compor um novo dado. Ainda no exemplo, a operação **E bit a bit** entre os números 1 e 2 representa a operação lógica **and** nos valores de 0001 e 0010. Desta forma, o primeiro bit do resultado será 0 and 0, o segundo 0 and 0, o terceiro 0 and 1 e o último 1 and 0. Resultando portanto, em 0000, no número 0.

A operação ou exclusivo, também conhecida como **XOR** equivale a:
a XOR b = ((NOT a) AND b) OR (a AND (NOT b))
a ^ b = (~a & b) | (a & ~b)

Cada operação de deslocamento de um bit, como a base é binária, equivale a multiplicação ou divisão por 2. Por exemplo, o número 4, representado por 0100 deslocado de um bit à direita se torna o número 2, representado por 0010.

Exemplo:

```
>>> 1 & 2
0
>>> 2^1
3
>>> (~2 & 1) | (2 & ~1)
3
>>> 8>>1
4
>>> 2<<3
16
```

Tabela 5.5.1: *Operadores*

Tipo	Operador	Descrição
Bit a bit	&	E
Bit a bit	|	Ou
Bit a bit	^	Ou exclusivo
Bit a bit	~	Complemento de um
Bit a bit	«	Deslocamento à esquerda
Bit a bit	»	Deslocamento à direita

5.6 Filiação e Identidade

São dois operadores, um de filiação e o outro de identidade. O de filiação é o operador *in*, que retorna verdadeiro se o valor está presente no outro. O segundo, o de identidade retorna verdadeiro se o lado esquerdo e direito da expressão apontam para o mesmo objeto, isto é, são iguais.

Exemplo:

```
>>> x = 1
>>> x in [0, 1, 2]
True
>>> x in [0, 2, 3, 4]
False
>>> y = 1
>>> x is y
True
>>> x is 1
True
```

Tabela 5.6.1: *Operadores*

Tipo	Operador	Descrição
Filiação	in	Está em
Identidade	is	Apontam para o mesmo objeto

5.7 Precedência

Expressões com mais de um operador são avaliadas segundo a precedência deles, similar a expressões matemáticas que também têm precedências. Na dúvida, deve-se usar os parênteses, pois são os de maior precedência.

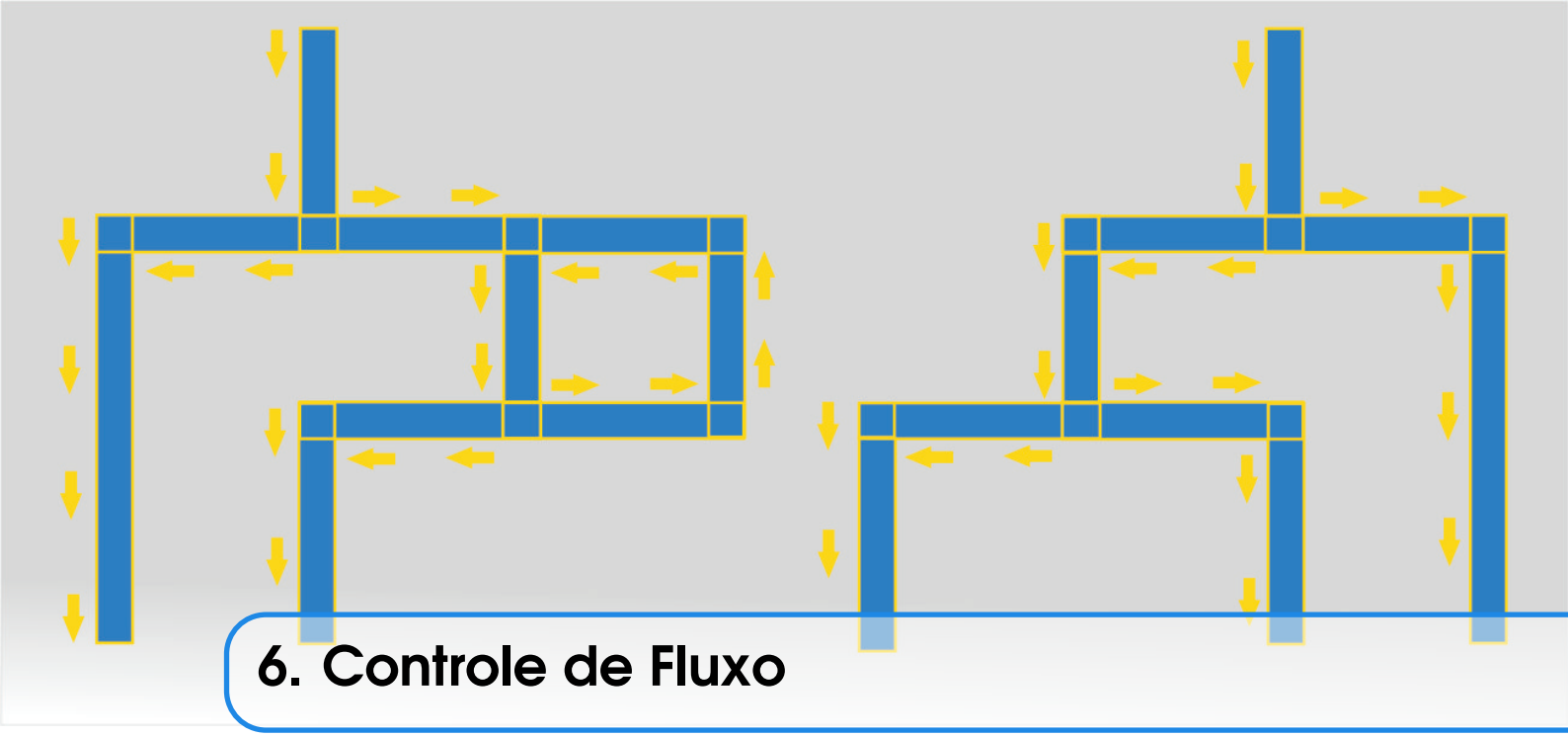
- ()
  Parênteses
- **
  Potenciação
- ~ + -
  Complemento, positivo e negativo (unários)
- * / % //
  Multiplicação e divisão
- + -
  Adição e subtração
- » «
  Deslocamento binário
- &
  E binário
- ^ |
  Não binário e ou binários
- <= < > >=
  Maior e menor que
- <> == !=
  Igual a e diferente de
- = %= /= //= -= += *= **=
  Atribuição
- is
  Identidade
- in
  Filiação
- not or and
  Relacionais

Exercícios

5.1 Crie um Python Script que receba um número inteiro n, correspondente a um valor em reais. Calcule a quantidade mínima de notas que um banco deve fornecer para atingir o valor. Notas disponíveis: 100,00 reais; 50,00 reais; 10,00 reais; 1,00 real.

5.2 Faça um Python Script que receba um valor t referente a uma quantidade total em segundos. Calcule a quantas horas:minutos:segundos a quantidade total de segudos corresponde.

5.3 Faça um programa que leia dois valores numéricos e realize a soma, subtração, multiplicação e divisão deles.



6. Controle de Fluxo

O controle de fluxo é o controle da sequência de comandos executados pela rotina. Um código é sempre executado linha por linha, na ordem que as encontram. Desta forma, se existem somente procedimentos simples, o funcionamento da rotina é linear. No entanto, com o controle de fluxo, é possível fazer com que o código tome decisões ou execute alguns procedimentos repetidas vezes. Este capítulo abordará as estruturas responsáveis por fazer o controle de fluxo no Python.

Tais ações equivalem a dizer para o compilador instruções como "se isso, faça aquilo"ou "enquanto isso, execute aquilo".

No Python, o início de uma dessas estruturas é indicada pelas palavras reservadas e o uso de dois pontos. Os procedimentos subordinados às estruturas devem estar indentados com um recuo de *tab* em referência a linha que indicia o início da estrutura. A própria quebra de indentação indicará o fim da estrutura.

Caso o código de Python não esteja devidamente indentado, ele não irá funcionar corretamente.

Sendo assim, uma estrutura deve-se parecer com:

```
ESTRUTURA  <expressao>  :
    procedimento1
    procedimento2
    ...
```

Algumas estruturas podem ainda estar subordinadas a outras estruturas. A este processo é

chamado de **aninhamento de estruturas**.

6.1 Estruturas Condicionais

O primeiro conjunto de estruturas de controle de fluxo são as estruturas condicionais, as responsáveis por tomadas de decisão, que indicarão uns procedimentos ou outros dependendo da condição avaliada.

As mais populares das estruturas condicionais são *if-else* e *switch*. No Python, não há a estrutura *switch*.

6.1.1 if-else

A estrutura *if* avalia uma expressão, caso a expressão seja verdadeira, os procedimentos subordinados são executados. Não é obrigatório, mas em seguida, pode-se adicionar apenas o *else*, com outros procedimentos subordinados, que serão executados caso a condição do if seja falsa.

exemploif1.py

```
1 x = int(input('digite um valor: '))
2 if x<10:
3     print('x<10')
4 else:
5     print('x>=10')
```

Neste exemplo, dependendo do valor que o usuário atribuir a x, uma mensagem diferente será impressa.

Pode haver também um tipo de *else if*, no caso de haver mais de uma condição *senão*. Neste caso, deve-se usar a palavra reservada *elif*.

exemploif2.py

```
1 x = int(input('digite um valor: '))
2 if x>10:
3     print('x maior que 10')
4 elif x<=10 and x>=5:
5     print('x entre 5 e 10')
6 else:
7     print('x menor que 5')
```

Neste caso, um valor a x é atribuído pelo usuário. Se x for maior que 10, é impresso "x maior que 10". Senão e se x estiver entre 5 e 10 é impresso "x entre 5 e 10". Senão, isto é,

se nenhuma das outras condições forem verdadeiras, é impresso "x menor que 5".

6.2 Estruturas de Repetição

Já o segundo conjunto de estruturas, as estruturas de repetição, são responsáveis por criar laços que repetem os procedimentos a eles subordinados. A cada repetição é dada o nome de **iteração**.

6.2.1 while

O primeiro é o *while*. While avalia a condição a ele exposta a cada iteração, e enquanto ela for verdadeira, o laço é mantido. Quando a condição é falsa, o laço é terminado.

exemplowhile.py

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

Neste exemplo, o i começa valendo 0. O while avalia a condição que é, inicialmente, verdadeira, então a primeira iteração é realizada, que imprimirá 0 e incremetará o i. Isto acontecerá até que i seja igual a nove, pois assim será impresso 9, o i será incrementado para 10 e a condição para o while será falsa, assim o laço irá terminar.

6.2.2 for

O segundo é o *for*. Ao invés de avaliar uma condição, como o while, o for, no Python, atribui valores a uma variável até que a lista deles acabe. Esses valores podem ser de tipos diferentes e são atribuídos segundo a ordem que aparecem. Quando a lista de valores acaba, o loop termina. Para a atribuição é necessário o operador de filiação, *in*.

exemplofor1.py

```
1 for i in [0, 'a', 2, 5, 1]:
2     print(i)
```

Neste caso, serão impressos os valores, na ordem que aparecem: 0, a, 2, 5 e 1.

Quando os valores a serem atribuídos são números crescentes, pode-se usar o comando *range(X)*, sendo assim, serão feitas X interações, atribuindo a variável valores de 0 a X-1.

exemplofor2.py

```
1 for i in range(10):
2     print(i)
```

Neste caso, serão impressos os valores: 0, 1, 2, 3, 4, 5, 6, 7, 8 e 9.

6.3 Desvio Incondicional

Os desvios incondicionais são mudanças da execução do programa para outra linha, isto é, desviar o código para outra parte. Há dois comandos de desvio, **break** e **continue**.

6.3.1 break

O primeiro deles, o break, serva para sair do laço. Exemplos:

```
>>> for i in range(0, 200000):
     print(i)
     if i == 5:
        break
```

Neste caso, o for estará preparado para iterar a variável i de 0 até 199999, no entanto, caso i seja igual a 5, haverá um break, isto é, sairá do laço, Assim, só serão impressos os valores 0, 1, 2, 3, 4 e 5.

```
>>> from random import randint
>>> while True:
      x = randint(0, 10)
      print(x)
      if x == 5:
        break
```

Neste exemplo, *while True* indica um **loop infinito**, isto é, nunca sairá dele. No entanto, caso x seja igual a 5, o loop é quebrado pelo comando break. Sendo assim, este laço irá atribuir valores aleatórios e inteiros entre 0 e 10 para x até que seja atribuído o valor 5. Todos os valores atribuídos, até chegar no 5 serão impressos.

6.3.2 continue

O segundo desvio é o continue. Ele serve para, a partir do momento que é atingindo, pular para a próxima iteração do laço, ignorando os procedimentos abaixo dele. Exemplos:

```
>>> for x in range(0, 10):
  if x == 5:
    continue
  print(x)
```

Neste exemplo, o for irá iterar a variável x de 0 a 9 e imprimir seu valor. No entanto, caso a x tenha o valor de 5, irá pular para a próxima iteração, ignorando os procedimentos abaixo, ou seja, não o imprimindo. Neste caso, então, serão impressos todos os valores de 0 a 9, exceto o 5.

6.3.3 pass

Em um primeiro momento, este comando pode parecer sem utilidade, mas futuramente poderá ser útil, para ignorar exceções ou definir classes vazias, por exemplo.

A função do pass é indicar que o interpretador deve prosseguir normalmente, isto é, deixar passar.

Exercícios

6.1 Faça um código que receba um número n do usuário e imprima os n primeiros números da sequência de Fibonacci.

6.2 Calcule 9! usando a estrutura for.

6.3 Calcule 9! usando a estrutura while.

6.4 Faça um programa que liste para o usuário um menu com quatro opções, sendo cada uma referente à uma operação matemática básica. Após o usuário ter escolhido a opção, leia dois valores e realiza a operação selecionada.

6.5 Faça um programa que receba um número inteiro n e o fatore.

6.6 Em algumas linguagens há a estrutura condicional switch, que não está presente no Python. Qual alternativa poderia ser usada, então, para o seguinte trecho de código genérico?

```
switch (X){
    case 1:
        foo(10);
        break;
    case 2:
        foo2(X);
        break;
    case 3:
        foo(5);
        break;
    otherwise:
        print('Erro')
}
```

II Conceitos Avançados

7. Estrutura de Dados II

É possível compor estruturas de dados primitivas, apresentadas no capítulo Estrutura de Dados, e armazená-las em outras estruturas, chamadas de **estruturas compostas**. Este capítulo irá apresentar melhor domínio de variáveis e conceitos mais avançados ao tratar das estruturas compostas.

Assim como os outros tipos de dados, os comandos *type()* e *dir()* ainda valem para exibir o tipo e mostrar atributos e métodos, respectivamente.

7.1 Listas

Listas são cadeias de valores, isto é, armazenam mais de um valor. São estruturas **ordenadas**, **mutáveis** e **heterogêneas**. São ordenadas, pois cada valor tem seu índice, na ordem que estão armazenados. São mutáveis, pois é possível alterar os valores. E são heterogêneas, pois os valores podem ser de tipos diferentes.

Podem ser atribuídas usando colchetes:

```
>>> x = [10, 5, 11, 0, 3]
>>> x
[10, 5, 11, 0, 3]
```

DICA: Listas vazias podem ser criadas usando-se uma das duas sintaxes a seguir:
x = []
x = list()

Como dito anteriormente, a lista é ordenada. Cada um dos 5 elementos da lista acima possui um índice, na ordem em que estão armazenados, de 0 a 4. Um item pode ser selecionado individualmente pelo uso dos colchetes, *Lista[índice]*.

```
>>> x[0]
10
>>> x[4]
3
```

Como a lista é mutável, os valores nela armazenados podem ser alterados e novos valores podem ser adicionados.

```
>>> L = [1, 4.5, 'abc', 3 + 4j, [1, 2, 3, 4]]
>>> L[2] = 5
>>> L[4] = 5
>>> L
[1, 4.5, 5, (3+4j), 5]
```

Para a adição de novos valores pode-se usar o método *append(valor)* ou o operador "+", que no caso das listas, executa concatenação.

```
>>> A = [1, 2, 3]; B = [4, 5, 6]
>>> C = A + B
>>> C
[1, 2, 3, 4, 5, 6]
>>> C += [7] #equivale a C = C + [7]
>>> C
[1, 2, 3, 4, 5, 6, 7]
>>> C.append('abc')
>>> C
[1, 2, 3, 4, 5, 6, 7, 'abc']
```

A heterogeneidade permite que a lista armazene tipos variados, incluindo também outras listas:

```
>>> L = [ 1, 4.5, 'abc', 3 + 4j, [1, 2, 3, 4]]
>>> L
[1, 4.5, 'abc', (3+4j), [1, 2, 3, 4]]
>>> L[4]
[1, 2, 3, 4]
>>> L[4][0]
1
```

DICA: Índices negativos, a partir do -1, percorrem a lista de trás para frente.

Alguns operadores aritméticos e relacionais podem ser usados com listas, como já abordado acima o operador "+". Basicamente, podem ser concatenadas e são comparáveis. A concatenação pode ser feita pelo operador de soma ou múltiplas concatenações podem ser feitas com o operador de multiplicação, ainda podem ser usados operadores de atribuição para realizar esta ação e atribuir simultaneamente o resultado à variável. A comparação é feita a partir do primeiro elemento e na sequência até que seja possível determinar qual maior e qual menor ou se são iguais.

Tabela 7.1.1: *Operadores em Listas*

Tipo	Operador	Descrição
Aritmético	+	Concatenação
Aritmético	*	Múltiplas concatenações
Relacional	==	Igual a
Relacional	!=	Diferente de
Relacional	<>	Diferente de
Relacional	>	Maior que
Relacional	>=	Maior ou igual a
Relacional	<	Menor que
Relacional	<=	Menor ou igual a

7.1.1 Funções

O comando *dir(list)* retorna os atributos e métodos das listas.

```
>>> dir(list)
['__add__', '__class__', '__contains__', '__delattr__',
   '__delitem__', '__dir__', '__doc__', '__eq__',
   '__format__', '__ge__', '__getattribute__',
   '__getitem__', '__gt__', '__hash__', '__iadd__',
```

```
'__imul__', '__init__', '__init_subclass__',
'__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__',
'__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'clear', 'copy', 'count',
'extend', 'index', 'insert', 'pop', 'remove',
'reverse', 'sort']
```

Veja algumas funções (métodos) da estrutura composta lista e o que fazem:

- append() Adiciona elementos a lista:

```
>>> x = [1, 2, 3]
>>> x.append(4)
>>> x
[1, 2, 3, 4]
```

- clear() Esvazia a lista:

```
>>> x = [1, 2, 3]
>>> x.clear()
>>> x
[]
```

- count() Retorna quantas ocorrências há do argumento passado:

```
>>> x = [1, 2, 3, 3, 3, 3, 4]
>>> x.count(3)
4
```

- index() Retorna o índice da primeira ocorrência do argumento passado:

```
>>> x = [1, 2, 3, 3, 3, 3, 4]
>>> x.index(3)
2
```

- insert() Insere na posição especificada um elemento:

```
>>> x = ['a', 'c', 'd']
>>> x.insert(1, 'b')
>>> x
['a', 'b', 'c', 'd']
```

- pop() Remove e retorna o elemento da posição especificada:

```
>>> x = ['a', 'b', 'c', 'd']
>>> x.pop(1)
'b'
>>> x
['a', 'c', 'd']
```

- remove() Remove o elemento especificado:

```
>>> x = ['a', 'b', 'c']
>>> x.remove('c')
>>> x
['a', 'b']
```

- reverse() Inverte a ordem dos elementos:

```
>>> x = [5, 4, 3, 2, 1]
>>> x.reverse()
>>> x
[1, 2, 3, 4, 5]
```

- sort() Ordena:

```
>>> x = [10, 3, 3, 10, 10, 6, 6, 4, 8, 7]
>>> x.sort()
>>> x
[3, 3, 4, 6, 6, 7, 8, 10, 10, 10]
```

7.1.2 Laço for

Os conjuntos finitos de valores usados na estrutura de repetição for para iterar uma variável, também é uma lista.

```
>>> x = [ 10, 5, 11, 0, 3]
>>> for i in x:
  print(i)
```

Neste caso, a lista x é usada como a lista de valores no qual i será iterado. Assim, serão impressos os valores 10, 5, 11, 0 e 3.

No entanto, fora da iteração for, o operador in retorna verdadeiro se o i está presente

em x, ou falso caso contrário.

```
>>> i = 10
>>> i in x
True
>>> i = 100
>>> i in x
False
```

Quando uma lista é criada a partir de um laço for, é possível reescrever o procedimento de forma mais compacta.

```
>>> A=[3, 4, 5, 10]
>>> soma1 = list()
>>> for i in A:
  soma1 += [i+1] #soma1.append(i+1)


>>> soma1
[4, 5, 6, 11]
```

Neste exemplo, uma nova lista soma1 é criada a partir de A somando-se 1 a cada elemento.

A sintaxe da forma compacta pode ser expressa da seguinte forma:
NovaLista = [Elemento for Variavel in VelhaLista if Condicao]
No Exemplo anterior, seria:

```
>>> A=[3, 4, 5, 10]
>>> soma1 = [i+1 for i in A]
>>> soma1
[4, 5, 6, 11]
```

É possível também usar a forma compacta com condições:

```
>>> A = [0, 0, 5, 3, 2, 7, 4, 10, 4, 7]
>>> N = list()
>>> for i in A:
```

```
  if i%2 == 0:
    N += [i*i]


>>> N
[0, 0, 4, 16, 100, 16]
```

Esse exemplo atribui a N os valores de A ao quadrado, mas apenas os que são pares.

```
>>> A = [0, 0, 5, 3, 2, 7, 4, 10, 4, 7]
>>> N = [i*i for i in A if i%2 ==0]
>>> N
[0, 0, 4, 16, 100, 16]
```

7.1.3 Fatiamento

É possível acessar partes da lista por meio de fatiamentos. Para isso, há três parâmetros, separados por dois pontos:

```
>>> lista[ inicio: fim: passo]
```

Índices negativos continuam válidos, bem como o passo negativo, que indicará que o fatiamento será de trás para frente. O parâmetro de início é de onde o fatiamento irá começar, o de fim-1 é onde o fatiamento irá parar e o passo é de quantos em quantos elementos serão pegos. Exemplo:

```
>>> X = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
>>> X[0:2:1]
[1, 2]
>>> X[0:4:1]
[1, 2, 3, 4]
>>> X[0:4:2]
[1, 3]
```

Nesse exemplo, primeiro é selecionado do elemento 0 até o elemento 1, de 1 em 1, depois selecionado do 0 até o 3 de 1 em 1 e, por fim, de 0 até o 3 de 2 em 2.

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

Esse exemplo mostra o passo negativo, que indica que o fatiamento percorre a lista no sentido inverso. Primeiro, toda a lista foi selecionada, porém com passo -1, e, por fim, a lista foi selecionada a partir do elemento 4 e contada de 1 em 1 em direção ao elemento 0.

É possível também omitir os parâmetros. Por padrão, o início vale 0, o fim vale o índice do último elemento + 1 (tamanho da lista) e o passo vale 1.

```
>>> X = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
>>> X[:2:1]
[1, 2]
>>> X[:4]
[1, 2, 3, 4]
>>> X[::2]
[1, 3, 5, 7, 9]
>>> X[:] #equivale a X
[1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
```

Nesse exemplo, X foi fatiado primeiro de 0 a 2, com passo 1, depois em 0 a 4, também com passo 1, em seguida a lista toda, com passo 2 e, por fim, a lista toda.

7.2 Tuplas

As tuplas são similares as listas, pois são **ordenadas** e **heterogêneas**. No entanto, diferentemente das listas, as tuplas são **imutáveis**. Este fato faz com que as tuplas sejam mais compactas e eficazes em termos de memória e eficiência.

A sua atribuição é feita por meio de parênteses:

```
>>> x = ('a', 1, 2)
>>> type(x)
<class 'tuple'>
>>> x
('a', 1, 2)
```

DICA: Tuplas vazias podem ser criadas usando-se uma das duas sintaxes a seguir:
x = ()
x = tuple()
No entanto, como as tuplas são imutáveis, esse procedimento não é comum.

As tuplas são imutáveis, no entanto, como são heterogêneas, se em uma tupla houver listas, os item das listas podem ser alterados normalmente, pois, apesar da tupla, as listas são mutáveis:

```
>>> x = ([0, 1, 2], 2, 2)
>>> x[1]
2
>>> x[1] = 2
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    x[1] = 2
TypeError: 'tuple' object does not support item assignment

>>> x[0]
[0, 1, 2]
>>> x[0][0] = 'mudei'
>>> x
(['mudei', 1, 2], 2, 2)
```

Alguns operadores aritméticos e relacionais também se aplicam as tuplas, muito similar a aplicação em listas. Basicamente, elas podem ser concatenadas e são comparáveis.

```
>>> X = (1, 2, 3)
>>> Y = (1, 2, 4)
>>> Z = (4, 0, 0)
>>> X < Y
True
>>> X < Z
True
>>> X + Y
(1, 2, 3, 1, 2, 4)
>>> Z*4
(4, 0, 0, 4, 0, 0, 4, 0, 0, 4, 0, 0)
>>> X += (4,)
>>> X
(1, 2, 3, 4)
```

As tuplas são imutáveis, mas mesmo assim, pode-se realizar a operação de atribuição com concatenação, como no exemplo acima (X += (4,)). Isso é permitido, pois ao fazer esta operação os elementos da tupla não estão sendo alterados, o que ocorre é que uma nova tupla está sendo atribuída a variável. Ou seja, a regra da imutabilidade está preservada.

Tabela 7.2.1: *Operadores em Tuplas*

Tipo	Operador	Descrição
Aritmético	+	Concatenação
Aritmético	*	Múltiplas concatenações
Relacional	==	Igual a
Relacional	!=	Diferente de
Relacional	<>	Diferente de
Relacional	>	Maior) que
Relacional	>=	Maior ou igual a
Relacional	<	Menor que
Relacional	<=	Menor) ou igual a

OBS : As tuplas também podem ser usadas em laços for. Nestes casos, a variável a ser iterada irá receber elemento por elemento da tupla.

7.2.1 Funções

O comando *dir(tuple)* retorna os atributos e métodos das tuplas.

```
>>> dir(tuple)
['__add__', '__class__', '__contains__', '__delattr__',
   '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
   '__getattribute__', '__getitem__', '__getnewargs__',
   '__gt__', '__hash__', '__init__', '__init_subclass__',
   '__iter__', '__le__', '__len__', '__lt__', '__mul__',
   '__ne__', '__new__', '__reduce__', '__reduce_ex__',
   '__repr__', '__rmul__', '__setattr__', '__sizeof__',
   '__str__', '__subclasshook__', 'count', 'index']
```

Algumas funções(métodos) da estrutura composta tupla e o que fazem:

- count() Retorna quantas ocorrências há do argumento passado:

```
>>> x = (1, 2, 3, 3, 3, 3, 4)
>>> x.count(3)
4
```

- index() Retorna o índice da primeira ocorrência do argumento passado:

```
>>> x = (1, 2, 3, 3, 3, 3, 4)
>>> x.index(3)
2
```

7.3 Dicionários

Listas e tuplas são indexadas pela ordem dos elementos, isto é, são **ordenadas**. Já os dicionários, não são. Ao invés disso, nesta estrutura, cada elemento recebe uma etiqueta. Além disso, são **mutáveis** e **heterogêneas**. Esta estrutura é a mais poderosa do Python, permitindo manipulações complexas.

Sua atribuição é feita usando-se chaves e dando etiquetas aos valores:

```
>>> X = {'idade': 21, 'matricula': 10000, 'nome': 'Joao'}
>>> X
{'idade': 21, 'matricula': 10000, 'nome': 'Joao'}
```

Como dicionários não são ordenados, para acessar elementos individualmente, usa-se suas etiquetas:

```
>>> X = {'idade': 21, 'matricula': 10000, 'nome': 'Joao'}
>>> X['idade']
21
>>> X['nome']
'Joao'
```

DICA: Dicionários vazios podem ser criadas usando-se uma das duas sintaxes a seguir:
x = {}
x = dict()

7.3.1 Funções

O comando *dir(dict)* retorna os atributos e métodos dos dicionários.

```
>>> dir(dict)
['__class__', '__contains__', '__delattr__',
   '__delitem__', '__dir__', '__doc__', '__eq__',
   '__format__', '__ge__', '__getattribute__',
   '__getitem__', '__gt__', '__hash__', '__init__',
   '__init_subclass__', '__iter__', '__le__', '__len__',
   '__lt__', '__ne__', '__new__', '__reduce__',
   '__reduce_ex__', '__repr__', '__setattr__',
   '__setitem__', '__sizeof__', '__str__',
   '__subclasshook__', 'clear', 'copy', 'fromkeys', 'get',
   'items', 'keys', 'pop', 'popitem', 'setdefault',
   'update', 'values']
```

Veja algumas funções(métodos) da estrutura composta dicionário e o que fazem:

- clear() Esvazia o dicionário:

```
>>> d = {'nome': 'joao', 'idade': 20}
>>> d.clear()
>>> d
{}
```

- get() Retorna o valor da chave passada como argumento. Caso haja um segundo argumento, será a resposta *default*, isto é, caso a chave não exista no dicionário, irá retornar o default.

```
>>> d = {'nome': 'joao', 'idade': 20}
>>> d.get('nome')
'joao'
>>> d.get('teste')
>>> d.get('teste', 'nao encontrado')
'nao encontrado'
```

7.3.2 Laço for

Dicionários também podem ser usados para iterar variáveis no laço for.

Caso seja utilizado apenas uma variável para ser iterada, a variável irá receber as chaves.

```
>>> d = {'nome': 'Joao', 'idade': 20, 'matricula': 1}
>>> for i in d:
  print(i)


nome
idade
matricula
```

Caso sejam utilizadas duas variáveis e usado o método items, a primeira receberá a chave e a segunda o valor:

```
>>> d = {'nome': 'Joao', 'idade': 20, 'matricula': 1}
>>> for i,j in d.items():
  print('i = ', i, 'j = ', j)


i =  nome j =  Joao
i =  idade j =  20
i =  matricula j =  1
```

7.4 Strings

As strings foram abordadas anteriormente como tipos primitivos de dados, mas são também cadeias de caracteres. Portanto, podem ser pensadas como uma estrutura composta. São sequências **ordenadas**, **imutáveis** e **homogêneas**. Ordenadas, pois cada elemento, isto é, cada letra possui seu índice, na ordem que estão armazenados. Imutáveis, pois não podem ser alteradas. Homogêneas, pois só aceitam um tipo de dado.

]

```
>>> P = 'Palavra'
>>> P[0]
'P'
>>> P[-1]
'a'
>>> P[5]
'r'
```

Também podem ser usadas para iterações em laços do tipo for.

```
>>> P = 'Palavra'
>>> for i in P:
      print(i)
```

Nesse caso, será impresso letra por letra de P, "Palavra".

O operador "+", ao ser usado em Strings, realiza concatenação, assim como em listas. No entanto, outros operadores também podem ser úteis:

```
>>> A = 'abc'; B = 'def'
>>> A + B
'abcdef'
>>> A < B
True
>>> print('_-' * 10 + '_')
_-_-_-_-_-_-_-_-_-_-_
```

Tabela 7.4.1: *Operadores em Strings*

Tipo	Operador	Descrição
Aritmético	+	Concatenação
Aritmético	*	Múltiplas concatenações
Relacional	==	Igual a
Relacional	!=	Diferente de
Relacional	<>	Diferente de
Relacional	>	Maior(alfabeticamente) que
Relacional	>=	Maior(alfabeticamente) ou igual a
Relacional	<	Menor(alfabeticamente) que
Relacional	<=	Menor(alfabeticamente) ou igual a

DICA: Os operadores relacionais, nos casos de menor ou maior comparam se uma string é alfabeticamente menor ou maior, isto é, se na ordem alfabética vem antes ou depois da outra.

7.4.1 Funções

O comando *dir(str)* retorna os atributos e métodos das listas.

```
>>> dir(str)
['__add__', '__class__', '__contains__', '__delattr__',
   '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
   '__getattribute__', '__getitem__', '__getnewargs__',
   '__gt__', '__hash__', '__init__', '__init_subclass__',
   '__iter__', '__le__', '__len__', '__lt__', '__mod__',
   '__mul__', '__ne__', '__new__', '__reduce__',
   '__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
   '__setattr__', '__sizeof__', '__str__',
   '__subclasshook__', 'capitalize', 'casefold', 'center',
   'count', 'encode', 'endswith', 'expandtabs', 'find',
   'format', 'format_map', 'index', 'isalnum', 'isalpha',
   'isascii', 'isdecimal', 'isdigit', 'isidentifier',
   'islower', 'isnumeric', 'isprintable', 'isspace',
   'istitle', 'isupper', 'join', 'ljust', 'lower',
   'lstrip', 'maketrans', 'partition', 'replace', 'rfind',
   'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
   'split', 'splitlines', 'startswith', 'strip',
   'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Veja algumas funções (métodos) da estrutura composta string e o que fazem:

- capitalize() Retorna a string com o primeiro caractere em maiúsculo e todos os caracteres em minúsculo:

```
>>> F = 'veja esta frase'
>>> F.capitalize()
'Veja esta frase'
```

- count() Retorna quantas ocorrências há do argumento passado:

```
>>> a = 'palavra'
>>> a.count('a')
3
```

- index() Retorna o índice da primeira ocorrência do argumento passado:

```
>>> a = 'palavra'
>>> a.index('a')
1
```

- isalnum() Retorna verdadeiro se todos os caracteres da frase forem alfanuméricos:

```
>>> A = 'abc11+'
>>> B = 'abc11'
>>> A.isalnum()
False
>>> B.isalnum()
True
```

- isalpha() Retorna verdadeiro se todos os caracteres da frase forem alfabéticos:

```
>>> A = 'abc11'
>>> B = 'abc'
>>> A.isalpha()
False
>>> B.isalpha()
True
```

- isdecimal() Retorna verdadeiro se todos os caracteres da frase representarem um número decimal:

```
>>> A = '10.9'
>>> B = '5'
>>> A.isdecimal()
False
>>> B.isdecimal()
True
```

- islower() Retorna verdadeiro se todos os caracteres da frase forem minúsculos:

```
>>> A = 'abCD'
>>> B = 'abcd'
>>> A.islower()
False
>>> B.islower()
True
```

- isnumeric() Retorna verdadeiro se todos os caracteres da frase forem numéricos:

```
>>> A = 'ab12'
>>> B = '1234'
>>> A.isnumeric()
False
>>> B.isnumeric()
True
```

- isupper() Retorna verdadeiro se todos os caracteres da frase forem minúsculos:

```
>>> A = 'abCD'
>>> B = 'ABCD'
>>> A.isupper()
False
>>> B.isupper()
True
```

- lower() Retorna a string com todos os caracteres em minúsculo:

```
>>> F = 'UMA FRASE'
>>> F.lower()
'uma frase'
```

- replace() Retorna o string trocando a primeira string do argumento pela segunda:

```
>>> F = 'Veja uma frase'
>>> F.replace(' ', '-')
'Veja-uma-frase'
>>> F.replace('uma', 'alguma')
'Veja alguma frase'
```

- split() Retorna a string separada de acordo com o separador, passado com argumento. Por padrão o separador é ' '.:

```
>>> F = 'Veja uma frase'
>>> F.split()
['Veja', 'uma', 'frase']
>>> F.split('a')
['Vej', ' um', ' fr', 'se']
```

- upper() Retorna a string com todos os caracteres em maiúsculo:

```
>>> F = 'uma frase'
>>> F.upper()
'UMA FRASE'
```

Exercícios

7.1 Crie um Python Script que conte quantas vezes cada nome está presente em uma lista ['nome1', 'nome2', ...] e armazena essa contagem em um dicionário {'nome1': xvezes, 'nome2': yvezes, ...}.

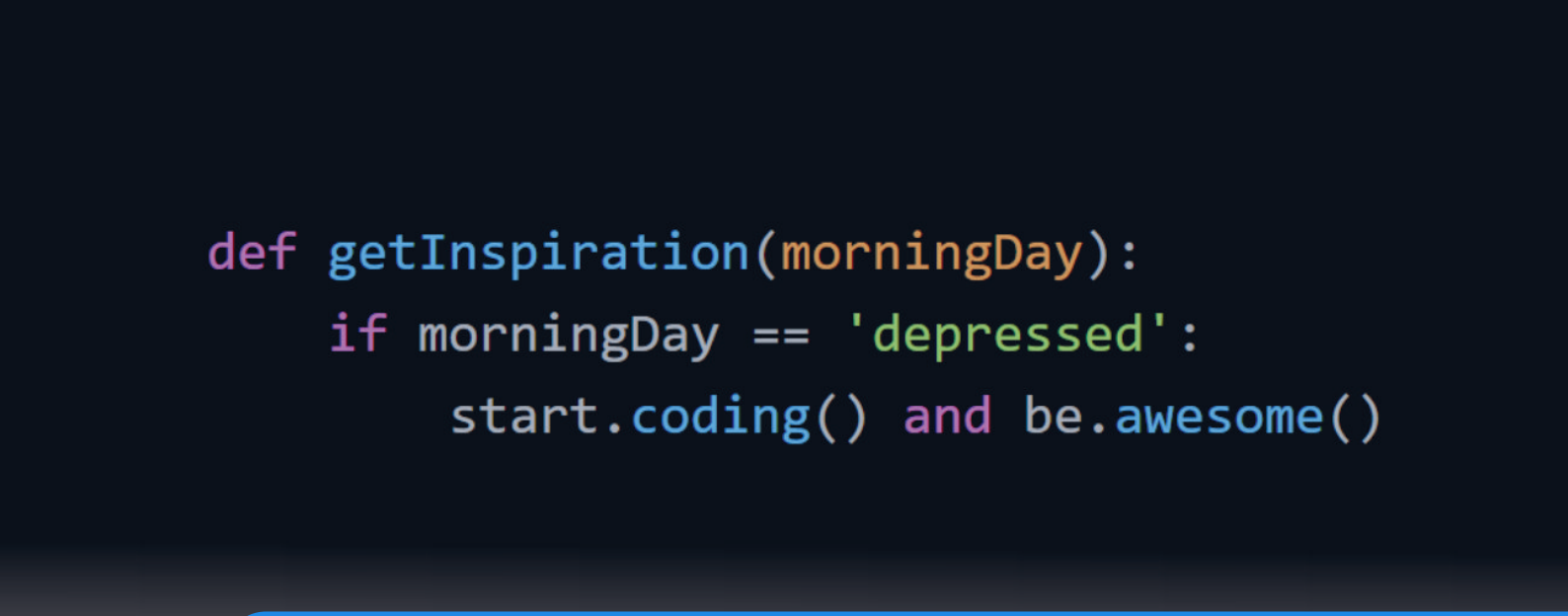
7.2 Crie um Python Script que realize o mesmo procedimento da questão anterior. No entanto, ao invés do conteúdo da lista nomes ser atribuído no próprio script, faça uma estrutura de repetição na qual ela leia uma string do usuário e adicione os nomes digitados por ele, um de cada vez, na lista nomes. O término da adição de nomes deve ser indicado quando o usuário inserir uma string vazia (”).

7.3 Crie um programa que lê uma mensagem do usuário. Com esta mensagem, faça uma nova omitindo trocando todos os caracteres de nomes próprios por ’*’. Exemplo: se a mensagem for ’Lucas foi ao shopping com Fernanda assistir aquele filme da Marvel’, a nova mensagem deverá ser ’***** foi ao shopping com ******** assistir aquele filme da ******’. Assuma que um nome próprio sempre começa com letra maiúscula e contém apenas letras.

7.4 Faça um programa que leia seis valores numéricos atribuindo-os à duas variáveis do tipo lista com três elementos cada. Cada variável irá representar um vetor, informe o produto escalar e o produto vetorial destes vetores.

7.5 Escreva um programa que leia um número N. Em sequência, ele deve ser N números e armazená-los em uma lista.

7.6 Com o programa acima, imprima: A soma dos itens, a média dos valores, o maior e o menor valor.



8. Comandos e Funções II

Este capítulo trará mais comandos e funções, agora, mais avançados, como definição de funções e leitura/escrita de arquivos utilizando Python.

8.1 Definição de Funções

Definições de funções auxiliam para automatizar procedimentos que são recorrentes no código, evitando a necessidade de escrevê-los várias vezes. Bem como as outras estruturas do Python, a estrutura de definição de funções também segue as regras da indentação e dos dois pontos.

Para definir a função, usa-se a palavra reservada *def*, seguida do nome da função, dos parâmetros entre parenteses e dos dois pontos. Os procedimentos devem ficar abaixo desta linha e indentados. Para retornar valores, ou apenas para indicar o término da função, deve-se usar a palavra *return*. Quaisquer estruturas de dados já vistas até agora podem ser retornadas.

Um simples exemplo que soma dois números deve se parecer com:

minhaprimeirafunc.py

```
1 #Partes inicias
2 #####
3
4 def minhafuncsoma(a, b):
5     s = a + b
```

```
6       return s
7
8   #Outras partes
9   ######
10
11  x = 2, y = 3
12  z = minhafunsoma(x, y)
13  # z sera 5
```

8.1.1 Escopo

No caso das definições de funções, alguns conceitos devem ser esclarecidos. Ainda no exemplo de **minhaprimeirafunc.py**, na chamada da função, linha 12, não foram usados a e b, foram usados x e y. Isso porque a e b são os parâmetros da função, isto é, ela comporta ser chamada com diferentes valores sendo passados no lugar de a e b.

Após a chamada, ao entrar na função, isto é, dentro do **escopo** da função, os valores passados no lugar de a e b são copiados para elas, que serão usadas para os procedimentos da função. No entanto, essas variáveis só existem no escopo da função, isto é, no resto do código, a e b ainda não foram declarados. Outro exemplo:

exemploescopo1.py

```
1   #Partes inicias
2   #####
3
4   def exemplo1():
5       A = 10
6       print(A)
7
8   #Outras partes
9   ######
10  exemplo1() #Chamando a funcao
11
12  print(A) # Tentando imprimir A
```

Neste exemplo, a primeira impressão será 10, mas a segunda impressão causará um erro, imprimindo algo como:

```
Traceback (most recent call last):
  File "C:/.../exemploescopo1.py", line 12, in <module>
    print(A)
NameError: name 'A' is not defined
```

Isso porque A só existe no escopo da função, ou seja, A é uma **variável local**.

As variáveis que podem ser acessadas em qualquer escopo, isto é, tem sua declaração hierarquicamente acima das outras, são chamadas de **variáveis globais**.

exemploescopo2.py

```
1  # Partes inicias
2  #####
3
4  def exemplo1():
5      #A = 10
6      print(A)
7
8  # Outras partes
9  ######
10 exemplo1()  # Chamando a funcao
11 A = 10
```

Nesse exemplo, A é uma variável global, mas sua declaração está após a chamada da função, portanto, a variável não existirá no escopo da função. Se as linhas 10 e 11 forem trocadas, como no exemplo 3, A será global e existirá também no escopo da função. Desta forma, será impresso 10.

exemploescopo3.py

```
1  # Partes inicias
2  #####
3
4  def exemplo1():
5      #A = 10
6      print(A)
7
8  # Outras partes
9  ######
10 A = 10
11 exemplo1()  # Chamando a funcao
```

Se uma variável local tiver o mesmo nome de uma variável global no escopo de uma função, a variável local irá prevalecer.

exemploescopo4.py

```
1  # Partes inicias
2  #####
3
```

```
4  def exemplo1():
5      A = 1
6      print(A)
7
8  # Outras partes
9  ######
10 A = 10
11 exemplo1()  # Chamando a funcao
```

Nesse exemplo, será impresso 1.

8.1.2 Recursividade

A recursividade é uma característica de certas funções que diz respeito a possibilidade de chamar a si mesmas. Nesses casos, ao definir uma função recursiva é importante atentar-se para impor uma condição que termine a recursão, porque caso contrário, a recursão será infinita e não retornará o resultado desejado. Para melhor compreensão, será desenvolvido um exemplo passo a passo. O objetivo é definir uma função recursiva que calcula o fatorial de um número n.

O primeiro passo é definir a função e estabeler sua propriedade recursiva. Nesta caso, pode-se definir uma função fat(n) que irá retornar n*fat(n-1), tornando a função recursiva:

funcaofat.py

```
1  def fat(n):
2          return n*fat(n-1)
3
4  n = int(input('Digite um numero: '))
5
6  r = fat(n)
7
8  print('n! = ', r)
```

No entanto, ainda não basta. Do modo anterior, ao se usar a função, suponha fat(4), o programa irá fazer 4*fat(3), depois 4*3*fat(2), depois 4*3*2*fat(1) e assim indefinidamente. É necessário impor uma condição de parada.

funcaofat.py

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

```
6  n = int(input('Digite um numero: '))
7
8  r = fat(n)
9
10 print('n! = ', r)
```

Agora, a recursividade será terminada quando n=1. Sabe-se também que fatorial de 0 é 1, e a função entrará em loop infinito novamente se for usada com fatorial de 0 ou números negativos. Para isso, melhores condições de parada devem ser impostas:

funcaofat.py

```
1  def fat(n):
2          if n < 0:
3                  print('Erro')
4                  return
5          elif n == 0:
6                  return 1
7          else:
8                  return n*fat(n-1)
9
10 n = int(input('Digite um numero: '))
11
12 r = fat(n)
13
14 print('n! = ', r)
```

Ainda pode-se argumentar que para números decimais a função causará outro loop infinito, pois não conseguirá chegar na restrição de n=0. As restrições e condições de parada dependem das condições da tarefa de cada programa e devem ser melhores definidas pelo programador em cada caso. O importante é ficar atento para não causar os erros citados acima.

8.2 Parâmetros Variados

Eventualmente, deseja-se fazer uma função com número de parâmetros variados, como a própria função print, por exemplo.

O primeiro modo é usar um asterisco no parâmetro, assim a quantidade será variável. Geralmente, usa-se o nome **args*.

Por exemplo:

```
1  >>> def foo(obrigatorio1, *args):
2    print('Foram passados', len(args)+1, 'parametros')
```

```
3
4
5 >>> foo(1)
6 Foram passados 1 parametros
7 >>> foo(2, list(), 3, 1+2j)
8 Foram passados 4 parametros
```

Nesse exemplo, números variados de parâmetros foram passados à função foo.

Uma outra maneira de passar um número variável de parâmetros, é fazê-la de forma similar a um dicionário, onde indica-se os parâmetros por palavras-chaves, como na função print, onde pode-se indiciar o argumento *sep*. A maneira de se fazer é usando dois asteriscos. Geralemente, o parâmetro é chamado de ***kwargs*, de *keyworded arguments*.

```
>>> def foo(**kwargs):
  if kwargs['op'] == '+':
    return kwargs['primeiro'] + kwargs['segundo']
  elif kwargs['op'] == '-':
    return kwargs['primeiro'] - kwargs['segundo']


>>> foo(primeiro = 2, segundo = 3, op = '+')
5
```

8.3 Leitura e Escrita de Arquivos

Ler e escrever arquivos por um programa pode ser muito útil. Seja para ler grandes quantidades de dados e tratá-los, seja para armazenar os resultados de determinada operação ou seja para automatizar ainda mais o trabalho realizado pelo programa.

Para iniciar o processo, seja de leitura ou de escrita, é necessário criar uma variável que será a identificadora do arquivo. Essa variável também pode ser chamada de *handle* do arquivo a ler lido/escrito. Sendo assim, precisa-se "abrir"o arquivo e atribuir o resultado à essa variável, como o exemplo a seguir:

```
>>> write_handle = open('new.txt', 'w')
```

A sintaxe da função open pode ser descrita como:

$$VariavelIdentificadora = open('arquivo','modo',encoding='codificacao')$$

Há um terceiro parâmetro, *encoding*, nele é especificado qual será a codificação do arquivo. Este parâmetro é opcional, por padrão seu valor é *'utf8'*.

O segundo parâmetro diz respeito ao modo como o arquivo será aberto, pode ser, essencialmente, de três tipos:

- 'r' de *read*, abre um arquivo existente para leitura. Caso o arquivo não exista, um erro é retornado.
- 'w' de *write*, cria um novo arquivo. Caso já exista um arquivo com o mesmo nome, ele é excluído para dar lugar ao novo.
- 'a' de *append*, abre um arquivo já existente para apensar, ou seja, adicionar conteúdo ou alterá-lo. Caso o arquivo não exista, um novo é criado.

Adicionalmente, pode-se adicionar o caractere de soma (+) para indicar que outras ações também podem ser feitas. Por exemplo, se um arquivo for aberto de modo 'w+', ele será criado e servirá para escrita. No entanto, adicionalmente, também poderá ser lido.

A variável criada é de uma classe especial e ela que será usada para a leitura ou escrita no arquivo. No exemplo acima, a variável seria da classe:

```
>>> type(write_handle)
<class '_io.TextIOWrapper'>
```

Como visto nos capítulos de Estruturas de Dados e Estruturas de Dados II, a classe de *write_handle* também possui atributos e métodos que podem ser visualizados pelo comando *dir()*:

```
>>> dir(write_handle)
['_CHUNK_SIZE', '__class__', '__del__', '__delattr__',
   '__dict__', '__dir__', '__doc__', '__enter__',
   '__eq__', '__exit__', '__format__', '__ge__',
   '__getattribute__', '__getstate__', '__gt__',
   '__hash__', '__init__', '__init_subclass__',
   '__iter__', '__le__', '__lt__', '__ne__', '__new__',
   '__next__', '__reduce__', '__reduce_ex__', '__repr__',
   '__setattr__', '__sizeof__', '__str__',
   '__subclasshook__', '_checkClosed', '_checkReadable',
   '_checkSeekable', '_checkWritable', '_finalizing',
   'buffer', 'close', 'closed', 'detach', 'encoding',
   'errors', 'fileno', 'flush', 'isatty',
   'line_buffering', 'mode', 'name', 'newlines', 'read',
   'readable', 'readline', 'readlines', 'reconfigure',
   'seek', 'seekable', 'tell', 'truncate', 'writable',
   'write', 'write_through', 'writelines']
```

Para escrever em um arquivo, pode-se usar o método *write()*. Caso o método seja usado mais de uma vez, a nova string será escrita imediatamente após a anterior. A escrita só será efetivada no arquivo após seu fechamento, realizado pelo método *close()*. Antes disso, o arquivo existirá, mas estará em branco. Adicionalmente, o método *writable()* retorna um booleano indicando se o arquivo pode ser escrito ou não. Exemplo:

```
>>> escreve = open('new.txt', 'w')
>>> escreve.writable()
True
>>> escreve.write('Escrevendo a primeira linha\n')
28
>>> escreve.write('Agora a segunda\n')
16
>>> escreve.close()
```

O valor retornado em cada escrita indica a nova posição que a variável escreve está apontando. Isto é, a cada escrita, a variável anda para frente posicionando-se imediatamente após o conteúdo escrito. É por isso que, a cada chamada do método de escrever, o conteúdo é escrito imediatamente após o anterior, porque a posição em que a variável estava apontando mudou.

Deste modo, um arquivo será criado (no diretório corrente) e conterá o seguinte conteúdo:

new.txt

```
Escrevendo a primeira linha
Agora a segunda
```

Tendo escrito esse arquivo, é possível lê-lo criando-se outra variável identificadora, agora de modo read. Similar ao método *writable()*, que indicava se era possível escrever no arquivo, há o método *readable()*, que indica se é possível ler o arquivo.

```
>>> ler = open('new.txt', 'r')
>>> ler.readable()
True
>>> ler.close()
```

Há alguns modos de se ler arquivos. O mais simples deles é usar o método *read()*, que retornará todo o conteúdo do arquivo como uma única string.

```
>>> ler = open('new.txt', 'r')
>>> ler.read()
'Escrevendo a primeira linha\nAgora a segunda\n'
>>> x = ler.read() #o conteudo ja foi lido
>>> x #Logo, x nao recebera nada
''
>>> ler.close()
>>> ler = open('new.txt', 'r')
>>> x = ler.read()
>>> x
'Escrevendo a primeira linha\nAgora a segunda\n'
>>> print(x)
Escrevendo a primeira linha
Agora a segunda

>>> ler.close()
```

Similarmente ao método *write()*, o método *read()* também muda a posição apontada pela variável. Sendo assim, ao usar o método pela primeira vez, a variável apontará para o final do arquivo. Sendo assim, a partir da segunda vez que o método for usado, será retornado vazio ”.

Um segundo modo de ler um arquivo é ler linha por linha usando a estrutura de repetição for. Neste modo, a uma nova variável será iterada na própria variável do arquivo. Assim, em cada iteração a nova variável irá receber uma linha do arquivo, e a variável identificadora irá apontar uma linha para frente. Os términos de linhas são demarcados pelo caractere quebra de linha $\backslash n$.

```
>>> ler = open('new.txt', 'r')
>>> for i in ler
SyntaxError: invalid syntax
>>> for i in ler:
  print(i, end='')


Escrevendo a primeira linha
Agora a segunda
>>> ler.close()
```

Para escrever no final de um arquivo já existente, sem apagá-lo, deve-se abrí-lo do modo ’a’, para apensar.

```
>>> a = open('new.txt', 'a')
```

```
>>> a.readable()
False
>>> a.writable()
True
>>> a.write('Finalmente, a terceira linha\n')
29
>>> a.close()
```

Deste modo, o arquivo será modificado com o novo conteúdo escrito em seu final:

new.txt

```
Escrevendo a primeira linha
Agora a segunda
Finalmente, a terceira linha
```

8.4 Modularização

A modularização é uma habilidade de programação muito importante, já abordada anteriormente. Com ela, é possível dividir os procedimentos de um longo código, melhorando assim a organização, visualização e depuração do programa.

Em Python, pode-se fazer a modularização facilmente criando módulos e pacotes. A rigor, um módulo é um arquivo *.py* que contém as descrições a serem usadas, e um pacote é uma junção de vários módulos em uma pasta. Um pacote pode ser popularmente chamado de biblioteca e pode, até mesmo, conter outros pacotes.

Para se criar um módulo, basta fazer um arquivo *.py* e para usá-lo, basta importá-lo com o comando import, como ilustra o exemplo:

modulo1.py

```
1 def foo():
2     print('Usando a funcao de um modulo')
3
4 def foo2():
5     pass
6
7 ...
```

principal.py

```
1 import modulo1
2
3 ...
```

```
4
5 modulo1.foo()
6
7 ..
```

Também pode-se usar a importação otimizada.

Para se criar um pacote, deve criar uma pasta e dentro desta pasta deve obrigatorimanete conter um arquivo chamado *__init__.py*, que indicará que se trata de um pacote. Feito isso, pode-se adicionar dentro da pasta do pacote os módulos e outros pacotes desejados.

Os módulos e pacotes devem ser salvos em um diretório corrente, assim só poderão ser importados se estiver nesse diretório, ou salvos dentro do diretório de instalação do Python, na pasta *Lib*, assim poderão ser importados sempre.

Exercícios

8.1 Defina uma função que recebe uma mensagem em string e um offset em inteiro. A função deverá fazer uma nova mensagem trocando cada caractere por outro caractere deslocado de uma quantidade igual ao offset. Por exemplo, se a mensagem for 'abcd' e o offset 1, a nova mensagem será 'bcde'. Por fim, a função deve escrever a nova mensagem em um arquivo .txt. Encripte somente caracteres alfabéticos.

8.2 Defina uma função que recebe uma mensagem encriptada e um offset e realize o procedimento contrário da função definida na quesão anterior. Esta função também deve escrever a função decriptada em um arquivo .txt.

8.3 Una as duas funções das questões anteriores em um único programa. Adicione um menu em que o usuário pode escolher encriptar ou decriptar uma mensagem.

8.4 Com o arquivo gerado na questão anterior, faça um módulo que pode ser importado nos programas.

8.5 Defina uma função multi_multiplicacao que receba um número variável de parâmetros. A função deve retornar a multiplicação de todos eles.

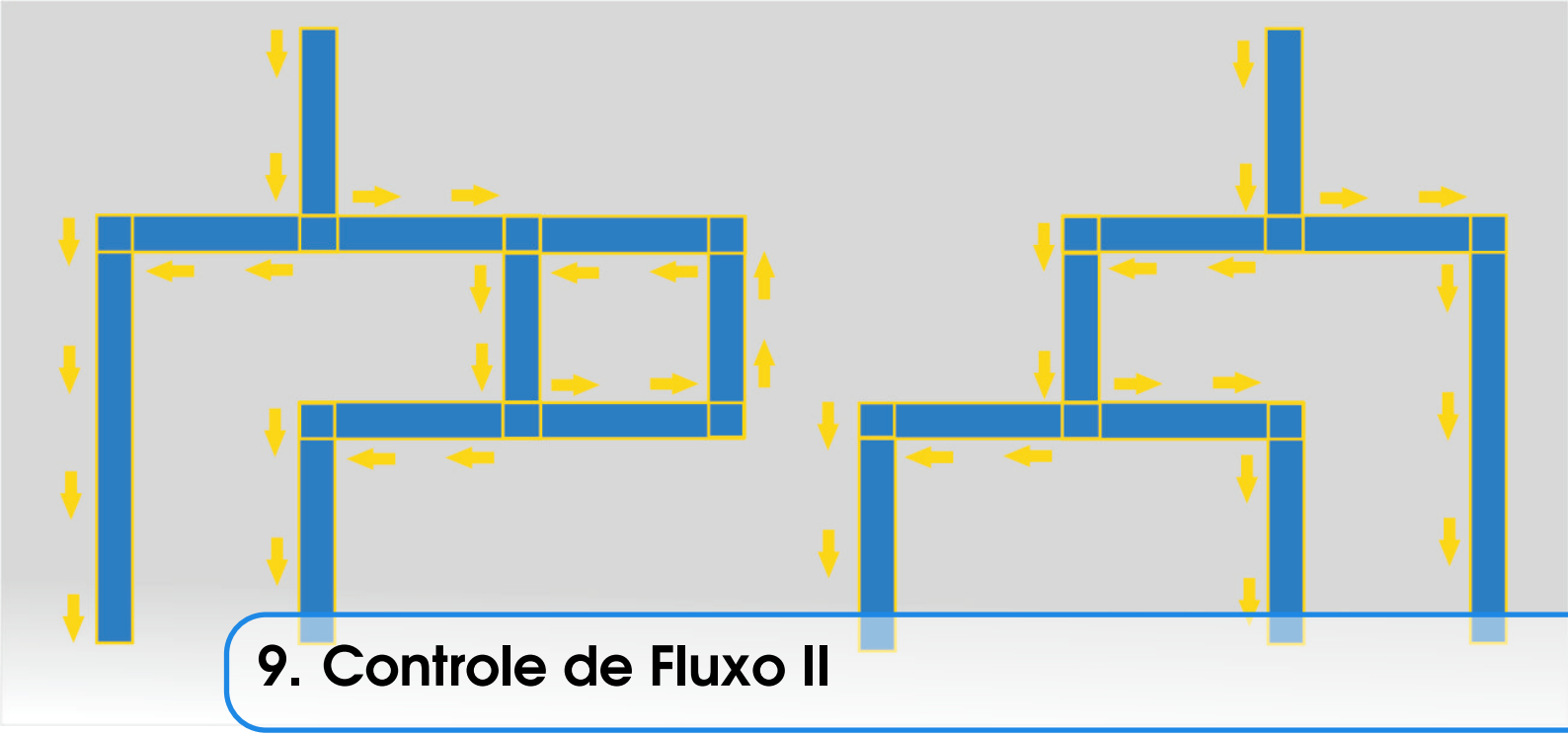
8.6 É possível definir uma função que recebe um número variado de argumentos do tipo *args e também do tipo **kwargs? Como o Python saberá qual argumento irá para args e qual para kwargs?

8.7 Dada a resposta do item anterior, suponha uma função foo() que recebe *args e **kwargs. É possível fazer a chamada da função com foo(1, 2, teste='test', 4) ? Por que?

8.8 É possível definir uma função que primeiro recebe argumentos **kwargs e depois recebe argumentos *args?

8.9 Crie uma função que calcule o enésimo número da sequência de Fibonacci recursivamente.

8.10 Crie um pacote para guardar todos os módulos criados durante o estudo dessa apostila.



9. Controle de Fluxo II

Tendo visto o controle de fluxo padrão, este capítulo irá abordar o tratamento de exceções, que são geradas por erros.

9.1 Exceções

As exceções são condições que realizam certas ações caso algo dê errado na execução do bloco corrente. O Python tem muitas exceções internas que fazem com que o programa retorne mensagens de erro, caso algum procedimento não seja executado propriamente.

Quando elas ocorrem, fazem com que o processo no bloco atual seja interrompido e uma exceção é lançada, isto é, o interpretador confere se existe alguma forma de tratar o erro no código, caso exista esse bloco será executado, caso contrário o programa irá *crashar*, isto é, ser desligado forçadamente.

Por exemplo:

um_codigo_qualquer.py

```
1 def funcao_C:
2     #procedimentos
3     #...
4
5 def funcao_B:
6     funcao_C
7     #procedimentos
8     #...
9
```

```
10 def funcao_A:
11     funcao_B
12     #procedimentos
13     #...
14
15 #...
```

Neste caso, uma determinada função A chama uma outra função B, que por sua vez chama uma função C. Se algum procedimento em C gerar um erro e uma exceção, ela tentará ser tratada em C, se não der, passará para B, se não der, passará para A, em cascata, até que seja tratada ou até gerar um *crash* no programa.

9.1.1 Tratamento

As exceções podem ser tratadas usando a estrutura *try*. A estrutura *try-except* é similar a estrutura condicional *if-else*. Dentro do bloco de try ficarão subordinados todos os procedimentos padrões que o programa deveria realizar. Após o bloco try, deve-se criar um bloco except, onde estarão os procedimentos do tratamento do erro.

Por exemplo:

exemplo_except01.py

```
1 Lista = ['a', 0, 1, 2]
2 for i in Lista:
3         x = 1/i
4         print('x =', x)
```

O código acima realiza a divisão de 1 por uma lista de valores. Obviamente, não será possível dividir 1 por 'a' e dividir 1 por 0, o que causará erros que leverão ao crash do programa, para tratar o erro, deve-se adicionar a estrutura try-except:

exemplo_except02.py

```
1  import sys #Modulo usado para informar o tipo do erro
2
3  Lista = ['a', 0, 1, 2]
4  for i in Lista:
5          try:
6                  x = 1/i
7                  print('x =', x)
8          except:
9                  print('Nao foi possivel dividir 1 por',
                       i)
10                 print(sys.exc_info()[0]) #Info sobre o
                       tipo do erro
```

Neste caso, o erro é tratado pelo programa, assim, mesmo que o procedimento dentro do try não seja possível, o except será executado e o bloco do laço for não gerará um crash. Será impresso: Nao foi possivel dividir 1 por a
<class 'TypeError'>
Nao foi possivel dividir 1 por 0
<class 'ZeroDivisionError'>
x = 1.0
x = 0.5

Nota-se ainda, que no exemplo acima a biblioteca *sys* foi importada e usado seu método *sys.exc_info()[0]* para informar o tipo de erro, no caso, *TypeError* e *ZeroDivisionError*, respectivamente.

O except pode ainda receber cláusulas, como se fossem condições. No exemplo anterior, os tipos de erros foram *TypeError* e *ZeroDivisionError*, que podem ser as cláusulas da exceção:

exemplo_except03.py

```
1 Lista = ['a', 0, 1, 2]
2 for i in Lista:
3         try:
4                 x = 1/i
5                 print('x =', x)
6         except TypeError:
7                 print('Nao foi possivel dividir numero
                      por caractere')
8         except ZeroDivisionError:
9                 print('Nao foi possivel dividir por
                      zero')
```

Neste caso, será impresso:

```
Nao foi possivel dividir numero por caractere
Nao foi possivel dividir por zero
x = 1.0
x = 0.5
```

9.1.2 Comando raise

Erros também podem ser sinalizadas pelo usuário pelo comando *raise*. Assim, o raise indicará que há um erro em seu bloco que esperará ser tratado.

exemplo_except04.py

```
try:
        a = int(input('Insira um valor inteiro
           positivo:\n'))
        if a <= 0:
                raise ValueError('ERRO: O valor deve ser
                   positivo')
except ValueError as erro:
        print(erro)
```

Execução do código acima:
Insira um valor inteiro positivo:
-2
ERRO: O valor deve ser positivo

Outra execução:
Insira um valor inteiro positivo:
10

9.1.3 Comando finally

A estrutura try também pode ser usada em conjunto com o *finally*. Os procedimentos subordinados ao finally irão ser executados independente do sucesso ou erro dos procedimentos subordinados ao try. Geralmente usado para liberar recursos externos, como por exemplo fechar um identificador de arquivo:

ex_finally.py

```
try:
        a = open('arquivo.txt', 'w')
finally:
        a.close()
```

9.1.4 Comando with

O código acima pode ainda ser escrito usando a estrutura *with*. Então, o mesmo procedimento usando with é ilustrado a seguir:

ex_with.py

```
with open('arquivo.txt', 'w') as a
```

O with, portanto, serve para finalizar os objetos inicializados, liberando os recursos externos. Geralmente, os programados lembram de fechar os identificadores de arquivos, o

with tem um uso mais geral.

A estrutura é da seguinte forma:

with_geral.py

```
#...
with expressão as variável:
    #bloco de operações do with
    #...
#...
```

A expressão é avaliada e deve corresponder a um objeto que contém os métodos *__enter__()* e *__exit__()*. O método de entrada é chamado antes da execução do bloco do with, que é executado em seguida. A expressão pode retornar um resultado que, se for o caso, é atribuído à variável, mesmo que não haja a operação de atribuição explícita ("=").

Após a execução do bloco do with, o método de saída do objeto é executado, mesmo que o bloco do with gere uma exceção.

10. Orientação a Objetos: Introdução

Tendo em vista a orientação a objetos em Python, este capítulo irá abordar alguns tópicos desse paradigma de programação. É importante ressaltar que os tipos de dados tratados até o momento são, na verdade, classes, sendo elas um conceito primordial desse tópico. Este capítulo poderia se chamar Estruturas de Dados III, já que será um aprofundamento no tema.

Para abordar todas as possibilidades e conceitos de orientação a objetos em Python, seria necessário um curso e uma apostila só para isso. Sendo assim, este capítulo irá apresentar os fundamentos de POO em Python.

Neste ponto, os chamados tipos de dados de Python (int, float, str, list, etc), agora são formalmente apresentados como **classes**. As classes facilitam a modularização e abstração, são elas que fazem do Python uma linguagem orientada a objetos.

10.1 Objetos

Um objeto é definido como uma estrutura de dados que é uma instância de uma **classe**. Isto é, um objeto é criado a partir de um molde, e este molde, chamados de classe. Pode-se fazer analogia com tipos de dados, por exemplo: uma variável a é do tipo inteira; um objeto a é da classe int. Classes são como tipos de dados, mas mais encorporados, e os objetos são as variáveis dessas classes.

Classe é definida como um agrupamento de valores e de operações.
Frequentemente classes diferentes possuem características comuns. As classes diferentes podem compartilhar valores comuns e podem executar as mesmas operações. Em Python tais relacionamentos são expressados usando derivação e herança.

Outro exemplo:
Se existir uma classe definida por um usuário *MinhaClasseCaderno*, então um objeto *Caderno1* pode ser criado da seguinte forma:

```
>>> Caderno1 = MinhaClasseCaderno()
```

Era exatamente este procedimento que era feito para:
se criar uma variável lista vazia *L = list()*)
ou para se converter "tipos de dados", agora apresentados como classes, *x = int(c)*.

No trecho de código a seguir, um objeto a é criado, e, como visto anteriormente, será do tipo int.

```
>>> a = 10
```

O que também poderia ser feito como:

```
>>> a = int(10)
```

Prosseguindo e fazendo um novo objeto a = b, a e b serão o mesmo objeto, o que poderia ser verificado pelo operador relacional "==", mas não é tão simples.

```
>>> a = 10
>>> b = 10
>>> a == b
True
```

O operador "=="verifica se os objetos possuem o mesmo valor, e não se são o mesmo objeto. Para essa verificação da forma correta, há o operador *is*.

```
>>> a = 2
>>> b = 2
>>> a is b
True
```

```
>>> a = int(2)
>>> b = float(2)
>>> a == b
True
>>> a is b
False
```

Isso ocorre pois, ao dizer que o objeto **a** recebe **b** significa que o objeto **a** recebe a referencia de **b**, ou seja, o mesmo endereço de memória, logo, são o mesmo objeto.

Uma classe possui instância de **atributos**, instância de **métodos** e **classes aninhadas**. Ou seja, um objeto irá possuir dentro dele alguns valores (atributos), algumas funções (métodos) e algumas classes (aninhadas).

A definição de uma classe é por meio da palavra reservada *class*, uma classe vazia pode ser definida como:

```
>>> class Vazia():
  pass

>>> X = Vazia()
```

DICA : Em Python 2, uma classe definida sem parâmetros é uma classe de um tipo antigo. Esse tipo, diferente dos novos, possuem uma quantidade menor de atributos e métodos, que pode ser visualizado pelo comando dir. Para que, em Python 2 uma classe seja do novo tipo, é necessário *object* como parâmetro.

10.2 Atributos

Os atributos são os valores que existem dentro do objeto. Por exemplo: uma classe Caderno será criada, definindo dois atributos, cor e np (número de págias):

```
>>> class Caderno():
  cor = 'preto'
  np = 10


>>> NovoCaderno = Caderno()
>>> NovoCaderno.cor
'preto'
```

```
>>> NovoCaderno.cor = "roxo"
>>> NovoCaderno.cor
'roxo'
```

No exemplo acima, uma classe é definida pelo usuário, todo obejto daquela classe terá dois atributos, cor valendo 'preto' e np valendo 10. Em seguida, uma variável (objeto) *NovoCaderno* da classe *Caderno* é criada. Em um primeiro momento, os atributos podem ser acessados pelo operador ponto ".", e seus valores também podem ser alterados.

```
>>> type(NovoCaderno)
<class '__main__.Caderno'>
>>> dir(NovoCaderno)
['__class__', '__delattr__', '__dict__', '__dir__',
   '__doc__', '__eq__', '__format__', '__ge__',
   '__getattribute__', '__gt__', '__hash__', '__init__',
   '__init_subclass__', '__le__', '__lt__', '__module__',
   '__ne__', '__new__', '__reduce__', '__reduce_ex__',
   '__repr__', '__setattr__', '__sizeof__', '__str__',
   '__subclasshook__', '__weakref__', 'cor', 'np']
```

Nota-se que as funções type e dir continuam valendo para as classes definidas pelo usuário.

O Python permite, a princípio, alterações no conteúdo de uma classe durante a sua execução. Pode-se, por exemplo, adicionar um novo atributo mesmo depois da classe já ter sido definida e instanciada em um objeto.

```
>>> class Vazio():
  pass

>>> X = Vazio()
>>> X.numero = 10
>>> X.numero
10
>>> dir(X)
['__class__', '__delattr__', '__dict__', '__dir__',
   '__doc__', '__eq__', '__format__', '__ge__',
   '__getattribute__', '__gt__', '__hash__', '__init__',
   '__init_subclass__', '__le__', '__lt__', '__module__',
   '__ne__', '__new__', '__reduce__', '__reduce_ex__',
   '__repr__', '__setattr__', '__sizeof__', '__str__',
   '__subclasshook__', '__weakref__', 'numero']
```

10.3 Métodos

Os métodos são funções intrínsecas aos objetos, que podem ser chamados de forma similar aos atributos e definidos como se fossem funções. Continuando com o exemplo da classe Caderno:

```
>>> class Caderno():
  cor = 'preto'
  np = 10
  def Descreva_se(self):
    print('Este objeto é um caderno, que contém cor e
       número de páginas')


>>> NovoCaderno = Caderno()
>>> NovoCaderno.Descreva_se()
Este objeto é um caderno, que contém cor e número de
   páginas
```

Todo método recebe como primeiro parâmetro o próprio objeto, e este parâmetro deve ser declarado explicitamente na definição do método, ele servirá para poder acessar os próprios métodos e atributos.

É importante ressaltar que self não é uma palavra reservada em Python, mas é amplamente usado por convenção e boas práticas de programação.

> **CURIOSIDADE** : Já foi questionado o porquê da necessidade de declarar o parâmetro self explicitamente, e não deixar a passagem automática, como em Java. Entretando, Guido van Rossun escreveu em seu blog o motivo de deixar a declaração explícita. `http://neopythonic.blogspot.com/2008/10/why-explicit-self-has-to-stay.html`

Existem alguns métodos especias que serão descritos a seguir.

10.3.1 Acessores, Modificadores e Deletores

São chamados de métodos acessores os métodos que acessam um atributo do objeto, mas não a modifiquem. Enquanto que os métodos modificadores acessam e modificam. Esses métodos também são chamados de *getters* e *setters*, respectivamente.

```
>>> class Caderno():
  cor = 'preto'
  np = 10
  def getCor(self):
    return self.cor
  def getNp(self):
    return self.np
  def setCor(self, novaCor):
    self.cor = novaCor
  def setNp(self, novoNp):
    self.np = novoNp


>>> C = Caderno()
>>> C.getCor()
'preto'
>>> C.setCor('amarelo')
>>> C.getCor()
'amarelo'
```

No exemplo acima, observa-se que existem dois métodos acessores, getCor e getNp e dois métodos modificadores setCor e setNp. No exemplo, também é possível observar o uso do primeiro parâmetro, o self.

Como visto anteriormente, a princípio, os atributos podem ser acessados e modificados diretamente pelo operador ponto ".", então esses dois tipos de métodos apresentados agora seriam inúteis. No entanto, mais adiante, o acesso e a modificação diretos de atributos serão proibidos, tanto por boas práticas de programação quanto por segurança. Classes mais complexas e modularizadas não permitem este tipo de operação. Sendo assim, esses dois tipos de métodos serão fundamentais para interagir com a classe e poderão ser melhores detalhados.

Já o método deletor, ou *deleter*, é o método capaz de deletar um atributo de uma classe.

```
>>> class Caderno():
  def getCor(self):
    return self.cor
  def getNp(self):
    return self.np
  def setCor(self, novaCor):
    self.cor = novaCor
  def setNp(self, novoNp):
    self.np = novoNp
  def delNp(self):
    del self.np
```

```
>>> C = Caderno()
>>> C.setNp(10)
>>> C.np
10
>>> C.delNp()
>>> C.np
Traceback (most recent call last):
  File "<pyshell#143>", line 1, in <module>
    C.np
AttributeError: 'Caderno' object has no attribute 'np'
```

10.3.2 Construtor

O método construtor é um método especial das classes. Ele é executado sempre que uma nova classe é iniciada e é denotado por *__init__*. Continuando com o exemplo:

```
>>> class Caderno():
  cor = 'preto'
  np = 10
  def __init__(self):
      print('Caderno iniciado')
  def getCor(self):
    return self.cor
  def getNp(self):
    return self.np
  def setCor(self, novaCor):
    self.cor = novaCor
  def setNp(self, novoNp):
    self.np = novoNp


>>> C = Caderno()
Caderno iniciado
```

O método também pode ser usado para dar valor aos atributos da classe, da seguinte forma:

```
>>> class Caderno():
  def __init__ (self, cor, np):
      print('Caderno iniciado')
    self.cor = cor
```

```
    self.np = np
  def getCor(self):
    return self.cor
  def getNp(self):
    return self.np
  def setCor(self, novaCor):
    self.cor = novaCor
  def setNp(self, novoNp):
    self.np = novoNp


>>> C = Caderno('azul', 100)
Caderno iniciado
>>> C.np
100
```

10.3.3 Representação

O método de representação também é um método especial de classes em Python. Ele serve para a exibição do resultado da impressão dos objetos dentro de funções do tipo print. Esse método é indicado por *__str__*.

```
>>> class Caderno():
  conteudo = ''
  def __init__ (self, cor, np):
    print('Caderno iniciado')
    self.cor = cor
    self.np = np
  def __str__ (self):
    return 'Caderno de %d páginas, da cor
       %s.\nConteúdo:\n%s' % (self.np, self.cor,
       self.conteudo)
  def escrever(self, msg):
    self.conteudo += msg
  def getCor(self):
    return self.cor
  def getNp(self):
    return self.np
  def setCor(self, novaCor):
    self.cor = novaCor
  def setNp(self, novoNp):
    self.np = novoNp


>>> C = Caderno('rosa', 50)
```

```
Caderno iniciado
>>> C.escrever('Ola mundo')
>>> print(C)
Caderno de 50 páginas, da cor rosa.
Conteúdo:
Ola mundo
```

A rigor, o que esse método faz não é definir como o objeto será impresso, mas sim como será convertido para o tipo str. Ao chamar a função print, ele será convertido para str e depois impresso.

10.3.4 Built-in

Alguns métodos podem representar algumas funções *built-in*, isto é, que já vem no Python. Por exemplo, o método *__float__* irá definir o que será retornado quando chamado *float(A)*. Alguns métodos como *__abs__* e *__str__* também são built-in, mas não serão listados a seguir porque já foram incluídos nas seções anteriores.

- object.__round__(self[, ndigits])
- object.__trunc__(self)
- object.__floor__(self)
- object.__ceil__(self)
- object.__complex__(self)
- object.__int__(self)
- object.__float__(self)

DICA : Mais informações sobre os métodos especiais que podem ser definidos em Python podem ser achadas na documentação: `https://docs.python.org/3/reference/datamodel.html`

10.3.5 Estáticos

Os métodos usados até agora são métodos dinâmicos. Isto é, para se usá-los, é preciso instanciar o objeto em uma classe. Por exemplo, se uma classe simples somador() for criado:

```
>>> class somador():
  def soma(self, a, b):
    return a + b
```

Para se usar o método soma, é preciso criar um objeto do tipo somador:

```
>>> S = somador()
>>> S.soma(2, 2)
4
>>> somador.soma(2, 2)
Traceback (most recent call last):
  File "<pyshell#41>", line 1, in <module>
    somador.soma(2, 2)
TypeError: soma() missing 1 required positional argument:
   'b'
```

Nota-se que não é possível usar o método diretamente pela classe, somente após instaciada em um objeto.

Sendo assim, há um tipo de método que pode ser chamado pela própria classe. Esse tipo é chamado como **método estático**. Seu funcionamento ficará parecido com o de uma função externa a classe, e é importante frisar que não é necessário (e nem permitido) passar o primeiro parâmetro referente a própria classe, geralmente chamado de self. Se usado, self não irá se referir a própria classe.

Um jeito prático de indicar que o método é estático é usar a tag *@staticmethod* antes de sua definição.

```
>>> class somador():
  def soma(self, a, b):
    return a + b
  @staticmethod
  def soma_estatica(a, b):
    return a + b


>>> somador.soma_estatica(2, 2)
4
>>> somador.soma(2, 2)
Traceback (most recent call last):
  File "<pyshell#58>", line 1, in <module>
    somador.soma(2, 2)
TypeError: soma() missing 1 required positional argument:
   'b'
>>> S = somador()
>>> S.soma(2, 2)
4
```

```
>>> S.soma_estatica(2, 2)
4
```

10.4 Sobrecarga de Operadores

As sobrecargas de operadores, a rigor, também são métodos especiais que podem ser definidos na classe. Com elas, é possível definir também o comportamento dos operadores com a classes.

```
>>>class Caderno():
  conteudo = ''
  def __init__ (self, cor, np):
    print('Caderno iniciado')
    self.cor = cor
    self.np = np
  def __str__ (self):
    return 'Caderno de %d páginas, da cor
       %s.\nConteúdo:\n%s' % (self.np, self.cor,
       self.conteudo)
  def __add__(self, other):
    return Caderno('branco', self.np + other.np)
  def escrever(self, msg):
    self.conteudo += msg
  def getCor(self):
    return self.cor
  def getNp(self):
    return self.np
  def setCor(self, novaCor):
    self.cor = novaCor
  def setNp(self, novoNp):
    self.np = novoNp

>>> A = Caderno('azul', 10)
Caderno iniciado
>>> B = Caderno('vermelho', 15)
Caderno iniciado
>>> X = A + B
Caderno iniciado
>>> print(X)
Caderno de 25 páginas, da cor branco.
Conteúdo:
```

No exemplo acima, é definida a operação que o operador *__add__* "+", irá realizar. Quando é feito A + B, é chamado o método A.__add__(B).

É possível definir a operação de vários operados, seguindo a mesma lógica:

+	object.__add__(self, other)
-	object.__sub__(self, other)
*	object.__mul__(self, other)
@	object.__matmul__(self, other)
/	object.__truediv__(self, other)
//	object.__floordiv__(self, other)
%	object.__mod__(self, other)
**	object.__pow__(self, other[, modulo])
<<	object.__lshift__(self, other)
>>	object.__rshift__(self, other)
&	object.__and__(self, other)
^	object.__xor__(self, other)
|	object.__or__(self, other)

Esses métodos são chamados para realizar operações seguindo a ordem dos operandos, por exemplo, *A / B* chama o método A.__truediv__(B), porque as operações foram definidas à esquerda. É possível definí-las à direita, adicionando um **r** no nome do método. Dessa forma, se *A / B* for chamado e o método __truediv__() não tiver sido implementado, o método B.__rtruediv__(A) é chamado, trocando os operandos.

Também é possível definir as operações de atribuição aritmética, adicionando um **i** antes do método. Por exemplo, se for executado *A += B*, será chamado o método A.__iadd__(B).

As operações unárias também seguem o mesmo processo, por exemplo, se chamado -*A* ou *A* é chamado o método A.__neg__() ou A.__invert__().

+	object.__pos__(self)
-	object.__neg__(self)
~	object.__invert__(self)

Os operadores relacionais também não ficam de fora:

<	object.__lt__(self, other)
<=	object.__le__(self, other)
==	object.__eq__(self, other)
!=	object.__ne__(self, other)
>	object.__gt__(self, other)
>=	object.__ge__(self, other)

10.5 Modificadores de Acesso

Outro conceito importante na orientação a objetos é o acesso. Isto é, a permissão de acessar atributos e métodos das classes. Geralmente, em linguagens clássicas de POO, usam-se três palavras reservadas para definir diferentes tipos de acesso. Elas são chamadas de **modificadoras de acesso**:

- *private*, privado. O acesso privado permite que os atributos e classes sejam acessados apenas dentro da própria classe.
- *public*, público. Os membros públicos podem ser acessados em qualquer lugar, até mesmo fora da classe.
- *protected*, protegido. Os membros protegidos podem ser acessados apenas dentro da classe e de suas sub-classes.

Em Python, não há estruturas e mecanismos que restrinjam o acesso. Há apenas uma convenção ao se dar um prefixo ao nome do método ou atributo com um ou dois caracteres underscore "_".

Por padrão, os atributos e métodos em Python são públicos. Os prefixados com um caractere underscore são protegidos e os com dois privados. De fato, apenas o privado não pode ser acessado diretamente, mas se a classe for instanciada em um objeto, poderá ser acessado usando-se a sintaxe: *objeto._class__privado*.

```
>>> class Acesso():
  public = 2
  _protected = 2
  __private = 2


>>> Acesso.public
2
```

```
>>> Acesso._protected
2
>>> Acesso.__private
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    Acesso.__private
AttributeError: type object 'Acesso' has no attribute
   '__private'
>>> X = Acesso()
>>> X.__private
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    X.__private
AttributeError: 'Acesso' object has no attribute
   '__private'
>>> X._Acesso__private
2
```

10.6 Property

property é uma classe. Ela define, para um atributo de uma classe, uma relação entre os métodos setters e getters. Por exemplo, uma classe do tipo pessoa possui um atributo idade. Se não utilizado os métodos getters e setters para verificações, qualquer valor pode ser atribuído a esse atributo:

```
>>> class Pessoa():
  def __init__ (self, idade):
    self.idade = idade


>>> P1 = Pessoa(15)
>>> P1.idade = 'abc'
>>> P1.idade
'abc'
```

Uma alternativa então, seria tornar o atributo privado e usar méotodos getters em setters com verificações:

```
>>> class Pessoa():
    __idade = ''
  def setIdade(self, idade):
    if type(idade) != type(int()):
```

```
      raise ValueError('Idade invalida')
    self.__idade = idade
  def getIdade(self, idade):
    return self.__idade


>>> P1 = Pessoa()
>>> P1.setIdade(10)
>>> P1.setIdade('abc')
Traceback (most recent call last):
  File "<pyshell#61>", line 1, in <module>
    P1.setIdade('abc')
  File "<pyshell#58>", line 4, in setIdade
    raise ValueError('Idade invalida')
ValueError: Idade invalida
```

No entanto, usar setters e getters não é intuitivo, e os métodos podem ter quaisquer nomes.

O property então liga os métodos ao atributo. Com ele, pode-se definir um método getter, setter e um método para deletar o atributo.

```
>>> class Pessoa():
  def setIdade(self, idade):
    print('O metodo setter foi chamado')
    if type(idade) != type(int()):
      raise ValueError('Idade invalida')
    self.__idade = idade
  def getIdade(self, idade):
    print('O metodo getter foi chamado')
    return self.__idade
  idade = property(getIdade, setIdade)


>>> P1 = Pessoa()
>>> P1.idade = 10
O metodo setter foi chamado
>>> P1.idade = ''
O metodo setter foi chamado
Traceback (most recent call last):
  File "<pyshell#77>", line 1, in <module>
    P1.idade = ''
  File "<pyshell#74>", line 5, in setIdade
    raise ValueError('Idade invalida')
ValueError: Idade invalida
```

Observa-se que ao tentar editar o valor diretamente, o método é chamado.

Sintaticamente, pode ser usado de várias formas:

```
# 1
idade = property(getIdade, setIDade)
# 2
idade = property(fget = getIdade, fset = setIdade)
# 3
idade = property()
idade = idade.getter(getIdade)
idade = idade.setter(setIdade)
```

E tudo isso também pode ser feito usando um método deletor, indicado por *fdel* ou *idade.deleter*.

10.7 Herança

A herança é o procedimento em que uma classe herda atributos e métodos de uma outra. A classe que herda é chamada de classe filha e a que passa de classe mãe.

Para fazer com que uma classe herde da outra, basta passá-la como parâmetro na definição da classe. Por exemplo, uma classe papel foi definida:

```
>>> class Papel():
        _conteudo = ''
        _tamanho = 200
        def escrever(self, mensagem):
                if len(self._conteudo + mensagem) <=
                   self._tamanho:
                        self._conteudo += mensagem
                else:
                        print('Nao cabe')
        def ler(self):
                return self._conteudo
```

Se for definida então, uma classe Caderno que depende da classe papel, todos os atributos e métodos (exceto os privados) serão diretamente herdados:

```
>>> class Caderno(Papel):
        def __init__(self, paginas):
                self._tamanho *= paginas

>>> A = Caderno(10)
>>> A._tamanho
2000
>>> A.escrever('aaa')
>>> A.ler()
'aaa'
```

Se houvessem métodos ou atributos privados, eles seriam herdados com o nome de *_Papel__privado*, como visto na seção de modificadores de acesso.

Se, dentro da classe filha, Caderno no exemplo acima, for redefinido um método ou atributo, ele sobrescreverá o herdado, prevalencendo, portanto, as novas definições.

CURIOSIDADE : A classe object é a mãe de todas as classes em Python.

Exercícios

Todos os exercícios deste capítulo cobrirão a construção, passo a passo, de uma classe com várias funcionalidades.

10.1 Defina uma classe chamada fracao, que contém dois atributos, num e den, inicializados com os valores que quiser.

10.2 Adicione a classe fracao o método de inicialização, em que os dois parâmetros passados na definição sejam referentes a num e den, respectivamente.

10.3 Adicione também o método especial de impressão, para que a classe seja mostrada do tipo "num/den".

10.4 Adicione a sobrecarga de operadores para realizar as quatro operações matemáticas básicas, de modo que, por exemplo, dois objetos A + B, resultem em uma terceira fracao. Não se preocupe em simplificar as frações

10.5 Adicione os métodos de conversão, isto é, se A for um objeto do tipo fracao, faça com que int(A) retorne um inteiro, float(A) retorne um float e bool(A) retorne um booleano. Utilize os critérios que julgar mais conveniente. Obs: str(A) já irá retornar uma string, pois este método foi definido na questão 10.3.

10.6 Adicione um método estático responsável por simplificar a uma fração qualquer, isto é, dado dois valor a e b, encontre a fração irredutível que represente a/b. Depois, adicione o método de simplificação no método construtor da classe.

10.7 Refine ainda mais a classe: faça com que a classe possa ser instanciada em um objeto com nenhum parâmetro, atribuindo 0/1, instanciada com apenas um parâmetro inteiro qualquer n, atribuindo n/1 à ela e melhorando a simplificação para que atribua, caso necessário, sinal negativo apenas para o numerador.

10.8 Altere os métodos de sobrecarga de operadores para que suportem também operações com inteiros, isto é, para que A + B seja válido, sendo A um objeto fracao e B um objeto int. Sugestões: utilize a função type para verificar a classe, utilize exceções.

10.9 Adicione os operadores unários, +, -, e a função abs à classe fracao.

10.10 Adicione os operadores relacionais à classe fracao.

Divirta-se com sua nova classe!

```
def getInspiration(morningDay):
    if morningDay == 'depressed':
        start.coding() and be.awesome()
```

11. Pacotes

Este capítulo irá citar algumas bibliotecas(pacotes) populares de Python com a finalidade de mostrar que a linguagem pode ser facilmente adaptada para uma aplicação e também de entreter o leitor.

Para aprofundar em cada biblioteca ou aplicação, seria necessário um curso específico sobre isso, pela enorme capacidade do Python e de seus pacotes. Este capítulo, portanto, é a título de curiosidade.

É importante ressaltar que no site oficial do Python existe a documentação dos milhares de projetos já lançados e é possível também criar um novo: `https://pypi.org/`.

11.1 Math

O pacote math já foi citado e usado anteriormente. Ele é *built-in*, isto é, já vem com o Python. Sua utilidade é notória, contém grande parte dos procedimentos e constantes matemáticas básicas.

Dentro deste pacote, pode-se citar as seguintes funções:

- math.ceil(x) Retorna o menor inteiro maior ou igual a x;
- math.floor(x) Retorna o maior inteiro menor ou igual a x;
- math.factorial(x) Retorna o fatorial de x se x for um valor inteiro não negativo;

- math.gcd(x, y) Retorna o maior divisor comum entre x e y, se x e y não forem zero;
- math.isfinite(x) Retorna verdadeiro se x não é infinito ou NaN (Not a Number);
- math.isinf(x) Retorna verdadeiro se x é infinito (positivo ou negativo);
- math.isnan(x) Retorna verdadeiro se x for NaN;
- math.exp(x) Retorna e exponencial de x;
- math.log(x [,y]) Se y não for passado, isto é, se houver apenas um argumento, retorna $ln(x)$, caso contrário, retorna $log_y x$;
- math.sin(x) Retorna o seno de x;
- math.cos(x) Retorna o cosseno de x;
- math.tan(x) Retorna a tangente de x;
- math.asin(x) Retorna o arco-seno de x;
- math.acos(x) Retorna o arco-cosseno de x;
- math.atan(x) Retorna o arco-tangente de x;
- math.sinh(x) Retorna o seno hiperbólico de x;
- math.cosh(x) Retorna o cosseno hiperbólico de x;
- math.tanh(x) Retorna a tangente hiperbólica de x;
- math.asinh(x) Retorna o arco-seno hiperbólico de x;
- math.acosh(x) Retorna o arco-cosseno hiperbólico de x;
- math.atanh(x) Retorna o arco-tangente hiperbólico de x;
- math.atan2(x) Retorna o arco-tangente de quatro quadrantes de x.

E as constantes:

- math.pi Retorna um valor aproximado de pi;
- math.e Retorna um valor aproximado para o número neperiano, ou exponencial de 1;
- math.inf Valor para representar infinito;
- math.nan Valor para represetnar NaN.

11.2 Random

11.3 Cores no Terminal

Existem muitas formas de fazê-lo. Uma delas é usando nas próprias strings que serão impressas, marcações especiais.

No caso de usuários de Windows, deve instalar o pacote colorama pelo terminal:

```
$  pip install colorama
```

Após isso, no terminal que for colorir deve-se usar o init() da biblioteca colorama.

```
>>> from colorama import init
>>> init()
```

Assim, o terminal do Windows aceitará as cores.

Pode-se digitar a marcação especial diretamente na string para que ela fique colorida, no entanto, ao invés de ter que memorizar códigos para estas cores, pode-se importar também da biblioteca colorama Fore, Back e Style, que já contém as marcações e apenas concatená-las com as strings.

```
>>> from colorama import Fore, Back, Style
>>> print(Fore.RED + 'some red text')
>>> print(Back.GREEN + 'and with a green background')
>>> print(Style.DIM + 'and in dim text')
>>> print(Style.RESET_ALL)
>>> print('back to normal now')
```

Os efeitos disponíveis são:

Fore: BLACK, RED, GREEN, YELLOW, BLUE, MAGENTA, CYAN, WHITE, RESET.
Back: BLACK, RED, GREEN, YELLOW, BLUE, MAGENTA, CYAN, WHITE, RESET.
Style: DIM, NORMAL, BRIGHT, RESET_ALL

A bibliteca *termcolor* também possui alguns atalhos para colorir o terminal.

Pode-se também, fazer uma classe com as marcações:

```
>>> class cores:
    HEADER = '\033[95m'
    OKBLUE = '\033[94m'
    OKGREEN = '\033[92m'
    WARNING = '\033[93m'
    FAIL = '\033[91m'
    ENDC = '\033[0m'
    BOLD = '\033[1m'
    UNDERLINE = '\033[4m'

>>> print(cores.WARNING + "Texto de aviso" + cores.ENDC)
```

11.4 NumPy

NumPy, de *Numeric Python*, é uma famosa e importante biblioteca de Python, conhecida também como uma biblioteca científica. É útil pela sua capacidade de processamento vetorial multidimensional homogêneo, estendendo o núcleo básico do Python.

CURIOSIDADE : Por convenção, a biblioteca NumPy é importada com o nome de np.
>>> import numpy as np

Vetores e matrizes podem ser criados pelo comando *array*, usando-se vírgulas e colchetes para identificá-los. Para selecionar elementos de um vetor ou matriz, deve-se usar também colchetes, muito similar ao uso e criação de listas:

```
>>> u = np.array([1, 0, 0])
>>> M = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> M
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> u
array([1, 0, 0])
>>> type(M), type(u)
(<class 'numpy.ndarray'>, <class 'numpy.ndarray'>)
>>> u[2]
0
>>> M[1, 2]
6
```

Pode-se especificar o tipo de dado com o parâmetro *dtype*, e há algumas funções especiais de criação de arranjos:

```
>>> np.eye(3)
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
>>> np.eye(3, dtype=np.int64)
array([[1, 0, 0],
       [0, 1, 0],
       [0, 0, 1]], dtype=int64)
>>> np.zeros((2, 3))
array([[0., 0., 0.],
       [0., 0., 0.]])
>>> np.ones((2, 3))
array([[1., 1., 1.],
       [1., 1., 1.]])
>>> np.full((2, 3), 5)
array([[5, 5, 5],
       [5, 5, 5]])
>>> np.random.random((2, 3))
array([[0.36596055, 0.55954819, 0.94322377],
       [0.53376681, 0.00523872, 0.05504161]])
```

Valores podem ser apensados em um arranjo pelo comando *append*:

```
>>> x = np.random.random((1, 2))
>>> x
array([[0.84375509, 0.04608968]])
>>> x = np.append(x, [1, 2])
>>> x
array([0.84375509, 0.04608968, 1.        , 2.        ])
```

DICA : Um arranjo vazio pode ser criado pela expressão:
np.array((0))

Operações matemáticas básicas podem ser feitas entre arrays ou entre array e número usando-se os operadores normalmente:

```
>>> x
array([0.84375509, 0.04608968, 1.        , 2.        ])
>>>
>>> x + 10
array([10.84375509, 10.04608968, 11.        , 12.        ])
>>> x - 10
array([-9.15624491, -9.95391032, -9.        , -8.        ])
>>> x * 5
array([ 4.21877543,  0.23044842,  5.        , 10.        ])
>>> x / 2
array([0.42187754, 0.02304484, 0.5       , 1.        ])
>>> y = np.array([1, 2, 3, 4])
>>> x + y
array([1.84375509, 2.04608968, 4.        , 6.        ])
>>> x / y
array([0.84375509, 0.02304484, 0.33333333, 0.5       ])
```

Para a multiplicação vetorial que representa o produto escalar entre dois vetores, há o método dot(). Para extrair o valor máximo, mínimo e a média há os métodos max(), min() e mean(), respectivamente:

```
>>> x
array([0.84375509, 0.04608968, 1.        , 2.        ])
>>> y
array([1, 2, 3, 4])
>>> x.dot(y)
11.935934454939305
>>> x.mean()
0.9724611925863914
>>> x.max()
2.0
>>> x.min()
0.04608968459373963
```

11.5 Matplotlib

Este é um pacote para plotar figuras de diversos estilos, como gráficos com linhas variadas, histogramas, gráficos de barras, dentre outros. É, geralmente, usado em conjunto com o NumPy, deixando o Python um pouco mais parecido com o Matlab.

Assim como a literatura convenciona a importação de NumPy como np, também convenciona a importação do módulo pyplot de matplotlib como plt.

Tendo os arranjos de x e y, pode-se plotar o gráfico usando:

```
>>> plt.plot(x, y)
>>> plt.show()
```

Como por exemplo, pode-se criar um script que plota alguns gráficos como este:

ExGraficos.py

```
import numpy as np
import matplotlib.pyplot as plt
import math
import colorama

"""
Estilos de linhas:
https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.plot.html#matplotlib.pyplot.plot
"""

def sine(x):
  return (np.sin(x))

def plot(x, y):
  plt.plot(x,y,'g-')
  plt.grid(True)

  # Limite dos eixos
  # plt.axis([0, 10*math.pi, -1, 1.01])

  # Nome dos eixos
  plt.ylabel('Eixo y')
  plt.xlabel('Eixo x')

  # Acentos devem ser usados com o codigo TeX
  # plt.title(r'$\ddot{o}\acute{e}\grave{e}\hat{O}\breve{i}\bar{A}\tilde{n}\vec{q}$', fontsize=20)

  # Titulo do grafico
  plt.title(r'$Gr\acute{a}fico$', fontsize = 15)


def cardiod():
  t = np.arange(0, 2*math.pi, 0.01)
  r = 1 + np.cos(t)
  x = r * np.cos(t)
  y = r * np.sin(t)
  plot(x,y)
```

```
38    plt.show()
39
40
41  def butterfly():
42    t = np.arange(0, 1000*math.pi, 0.01)
43    x = np.sin(t) * (np.exp(np.cos(t)) - 2*np.cos(4*t) -
         np.power(np.sin(t/12),5))
44    y = np.cos(t) * (np.exp(np.cos(t)) - 2*np.cos(4*t) -
         np.power(np.sin(t/12),5))
45    plot(x,y)
46    plt.show()
47
48  def tesla():
49    x1 = np.arange(0, 2.0*math.pi, 0.01)
50    x2 = np.arange(-2.0*math.pi/3.0, 4.0*math.pi/3.0, 0.01)
51    x3 = np.arange(-4.0*math.pi/3.0, 2.0*math.pi/3.0, 0.01)
52    y1 = np.sin(x1)
53    y2 = np.sin(x2 + 2.0*math.pi/3.0)
54    y3 = np.sin(x3 + 4.0*math.pi/3.0)
55
56    plot(x1,y1)
57    plot(x2,y2)
58    plot(x3,y3)
59    plt.show()
60
61  def lissajous_curve():
62    a = 4.0
63    b = 5.0
64    kx = 3.0
65    ky = 2.0
66    t = np.arange(0, 2.0*math.pi, 0.01)
67    x = a*np.cos(kx*t)
68    y = b*np.sin(ky*t)
69    plot(x,y)
70    plt.show()
71
72  def begin():
73    print('Curves: \n 1 - Cardiod \n 2 - Butterfly Curve \n
         3 - Tesla \n 4 - Lissajous Curve')
74    curve = int(input('Insert curve number: '))
75
76    if (curve == 1):
77      cardiod()
78    elif (curve == 2):
79      butterfly()
80    elif (curve == 3):
81      tesla()
82    elif (curve == 4):
```

```
83      lissajous_curve()
84    else:
85      print(colorama.Fore.RED + colorama.Back.WHITE +
          '\nInvalid input, try again')
86      print(colorama.Style.RESET_ALL)
87      begin()
88
89  if __name__ == '__main__':
90    # x = np.arange(0, 10*math.pi, 0.01)
91    # y = sine(x)
92    # y = np.arange(0)
93    # y = math.sin(x)
94    colorama.init()
95    begin()
```

IV

Índice

Resoluções

Estruturas de Dados

3.1 a) Tipo inteiro. Em Python: int.
b) Tipo booleano. Em Python: bool.
c) Tipo ponto flutuante. Em Python: float.
d) Tipo ponto flutuante. Em Python: float.
e) Tipo caractere. Em Python: str.
f) Tipo caractere. Em Python: str.

3.2 As variáveis a e b da questão são do tipo int, enquanto que as variáveis c e d são do tipo str. Sendo assim, o mesmo operador pode realizar diferentes operações com estruturas de dados diferentes, porque o Python é uma linguagem orientada a objetos. Neste caso, de fato realiza: a + b é uma soma de números inteiros e c + d é uma concatenação de strings.

3.3 Para que as operações tenham o mesmo resultado, pode-se converter os tipos de uma para outro:

```
>>> a = 1; b = 2; c = '1'; d = '2';
>>> a + b
3
>>> c + d
'12'
>>> str(a) + str(b)
'12'
>>> int(c) + int(d)
3
```

3.4 a) Permitido.
b) Permitido.
c) Não permitido, os nomes de variáveis devem começar com letras ou "_".
d) Permitido.
e) Permitido.
f) Não permitido, não é permitido espaço ou caracteres especiais.
g) Permitido.
h) Não permitido, a palavra é uma palavra reservada da linguagem.
i) Permitido.
j) Permitido.
k) Não permitido, a palavra é uma palavra reservada da linguagem.

Comandos e Funções

4.1 ex001.py

```
1  from math import pi, cos, sin, tan
2
3  Theta_deg = float(input('Insira o valor do
       angulo:\n'))
4  Theta_rad = Theta_deg * pi/180
5
6  print('Seno = ', sin(Theta_rad))
7  print('Cosseno = ', cos(Theta_rad))
8  print('Tangente = ', tan(Theta_rad))
```

4.2 ex002.py

```
1  from math import pi, cos, sin, tan
2  from numpy import deg2rad
3
4  Theta_deg = float(input('Insira o valor do
       angulo:\n'))
5  Theta_rad = deg2rad(Theta_deg)
6
7  print('Seno = ', sin(Theta_rad))
8  print('Cosseno = ', cos(Theta_rad))
9  print('Tangente = ', tan(Theta_rad))
```

Operadores

5.1 notasdedinheiro.py

```
    n = int(input('Digite o valor que deseja
       receber:\n'))

    nota_100 = n//100
    n = n%100
    nota_50 = n//50
    n = n%50
    nota_10 = n//10
    n = n%10
    nota_1 = n//1
    n = n%1

    print('Notas de 100,00 reais:', nota_100)
    print('Notas de 50,00 reais:', nota_50)
    print('Notas de 10,00 reais:', nota_10)
    print('Notas de 1,00 real:', nota_1)
```

5.2 calculotempo.py

```
    t = int(input('Digite o tempo total em
       segundos:\n'))

    segundos = t%60
    t //= 60
    minutos = t%60
    t //= 60
    horas = t

    print('O tempo digitado equivale a:\n',horas, ':',
       minutos, ':', segundos, sep='')
```

5.3 calculadora1.0.py

```
print('-'*10)
print('Bem vindo a Calculadora 1.0')
print('-'*10)

a = float(input('digite o valor de a\na = '))
b = float(input('digite o valor de b\nb = '))

print('a + b =', a+b)
print('a - b =', a-b)
print('a * b =', a*b)
print('a / b =', a/b)
```

6.1 Python Script
calculofib.py

```
n = int(input('Ate qual numero da sequencia de
   fibonacci deseja ver?\n'))
f = 0
prev_1 = 1
prev_2 = 0
print('Indice 1 f = 1')
for i in range(2, n+1):
     f = prev_1 + prev_2
     prev_2 = prev_1
     prev_1 = f
     print('Indice', i, 'f =', f)
```

6.2 Python Interativo

```
>>> f = 1
>>> for i in range(1, 10):
  f *= i


>>> print(f)
362880
```

6.3 Python Interativo

```
>>> f = 1; i = 1
>>> while(i<=9):
  f *= i
  i += 1


>>> print(f)
362880
```

6.4 Python Script
calculadora2.0.py

```
print('-'*10)
print('Bem vindo a Calculadora 2.0')
print('-'*10)

```

```
5  print('Escolha uma operacao:\n1 - soma;\n2 -
       subtracao;\n3 - multiplicacao;\n4 - divisao.')
6
7  Op = int(input('Operacao: '))
8
9  a = float(input('digite o valor de a\na = '))
10 b = float(input('digite o valor de b\nb = '))
11
12 if Op == 1:
13           print('a + b =', a+b)
14 elif Op == 2:
15           print('a - b =', a-b)
16 elif Op == 3:
17           print('a * b =', a*b)
18 elif Op == 4:
19           print('a / b =', a/b)
20 else:
21           print('Operacao invalida')
```

6.5 fatora.py

```
1  n = int(input('Insira o número que deseja fatorar\n'))
2
3  i = 2
4  while i<=n:
5           if n%i == 0:
6                    print(i, end = ' ')
7                    n //= i
8           else:
9                    i += 1
```

6.6 A alternativa é usar if-elses.

```
if X==1:
    foo(10)
elif X==2:
    foo2(X)
elif X==3:
    foo(5)
else:
    print('Erro')
```

Estruturas de Dados II

7.1 contanomes.py

```
    #Conta quantas vezes os nomes aparecem
    conta = dict()
    nomes = list()
    nomes = ['Joao', 'Arthur', 'Ary', 'Isabela',
       'Israel', 'Vitor', 'Joao']
    for nome in nomes:
        if nome in conta:
            conta[nome] += 1
        else
            conta[nome] = 1
    print(conta)
```

7.2 contanomes2.py

```
    #Conta quantas vezes os nomes aparecem
    conta = dict()
    nomes = list()

    a = ' '
    while a != '':
             a = input('Inserir mais um nome?\n')
             nomes += [a.capitalize()]

    nomes.remove('')
    #nomes = ['Joao', 'Arthur', 'Ary', 'Isabela',
       'Israel', 'Vitor', 'Joao']
    for nome in nomes:
        if nome in conta:
            conta[nome] += 1
        else:
            conta[nome] = 1
    print(conta)
```

7.3 TrocaNomeProprio.py

```
msg = input('Digite a mensagem:\n')

nova_msg = str()

for i in msg:
        if i.isupper():
                estou_em_nome = True
                nova_msg += '*'
                continue
        if estou_em_nome:
```

```
11                      if i.isalpha():
12                              nova_msg += '*'
13                              continue
14                      else:
15                              nova_msg += i
16                              estou_em_nome = False
17              else:
18                      nova_msg += i
19
20  print(nova_msg)
```

7.4 Produto_Escalar_Vetorial.py

```
X_str =input('Informe o vetor X:\n')
X_str_list = X_str.split()
X = [int(i) for i in X_str_list]

Y_str =input('Informe o vetor Y:\n')
Y_str_list = Y_str.split()
Y = [int(i) for i in Y_str_list]

produto_escalar = 0

for i in range(3):
        produto_escalar += X[i]*Y[i]

produto_vetorial = list()
produto_vetorial.append(  X[1]*Y[2] - X[2]*Y[1] )
produto_vetorial.append(  X[0]*Y[2] - X[2]*Y[0] )
produto_vetorial.append(  X[0]*Y[1] - X[1]*Y[0] )

print('Produto escalar =', produto_escalar)
print('Produto vetorial =', produto_vetorial)
```

7.5 Ex7_5.py

```
1  N = int(input('Insira o número N:\n'))
2
3  L = list()
4
5  for i in range(N):
6          x = float(input('Insira um número para a
               lista:\n'))
7          L.append(x)
8
9  print(L)
```

7.6 Ex7_6.py

```
    N = int(input('Insira o número N:\n'))

L = list()

for i in range(N):
        x = float(input('Insira um número para a
            lista:\n'))
        L.append(x)

print(L)

soma = 0
min = L[0]
max = L[0]

for i in L:
        soma += i
        if min > i:
                min = i
        if max < i:
                max = i

print('Soma = ', soma)
print('Média = ', soma/N)
print('Maior = ', max)
print('Menor = ', min)
```

Comandos e Funções II

8.1 encriptador.py

```
def encripte_msg(mensagem, offset):
    nova_msg = str()
    arquivo = 'mensagem_encriptada.txt'

    with open(arquivo, 'w') as arquivo_encriptado:
        for i in mensagem.lower():
            if i in alfabeto:
                index = alfabeto.find(i)
                nova_msg += alfabeto[(index - offset)
                    % 26]
            else:
                nova_msg += i
```

```
12
13          arquivo_encriptado.write(nova_msg)
14
15      print('A mensagem foi encriptada:', nova_msg, 'E
            escrita no arquivo', arquivo, sep='\n')
16
17  alfabeto = 'abcdefghijklmnopqrstuvwxyz'
18  msg = input('Digite a mensagem que deseja
        encriptar:\n')
19  offset = int(input('Digite o offset:\n'))
20  encripte_msg(msg, offset)
```

8.2 decriptador.py

```
1   def decripte_msg(mensagem, offset):
2       nova_msg = str()
3       arquivo = 'mensagem_decriptada.txt'
4
5       with open(arquivo, 'w') as arquivo_decriptado:
6           for i in mensagem.lower():
7               if i in alfabeto:
8                   index = alfabeto.find(i)
9                   nova_msg += alfabeto[(index - offset)
                        % 26]
10              else:
11                  nova_msg += i
12
13          arquivo_decriptado.write(nova_msg)
14
15      print('A mensagem foi decriptada:', nova_msg, 'E
            escrita no arquivo', arquivo, sep='\n')
16
17  alfabeto = 'abcdefghijklmnopqrstuvwxyz'
18  msg = input('Digite a mensagem que deseja
        decriptar:\n')
19  offset = int(input('Digite o offset:\n'))
20  decripte_msg(msg, offset)
```

8.3 cipher.py

```
1   def encripte_msg(mensagem, offset):
2       nova_msg = str()
3       arquivo = 'mensagem_encriptada.txt'
4
5       with open(arquivo, 'w') as arquivo_encriptado:
6           for i in mensagem.lower():
7               if i in alfabeto:
```

```
8                   index = alfabeto.find(i)
9                   nova_msg += alfabeto[(index + offset)
                       % 26]
10              else:
11                  nova_msg += i
12
13          arquivo_encriptado.write(nova_msg)
14
15      print('A mensagem foi encriptada:', nova_msg, 'E
           escrita no arquivo', arquivo, sep='\n')
16
17  def decripte_msg(mensagem, offset):
18      nova_msg = str()
19      arquivo = 'mensagem_decriptada.txt'
20
21      with open(arquivo, 'w') as arquivo_decriptado:
22          for i in mensagem.lower():
23              if i in alfabeto:
24                  index = alfabeto.find(i)
25                  nova_msg += alfabeto[(index - offset)
                       % 26]
26              else:
27                  nova_msg += i
28
29          arquivo_decriptado.write(nova_msg)
30
31      print('A mensagem foi decriptada:', nova_msg, 'E
           escrita no arquivo', arquivo, sep='\n')
32
33  alfabeto = 'abcdefghijklmnopqrstuvwxyz'
34  msg = input('Digite a mensagem que deseja encriptar ou
       decriptar:\n')
35  offset = int(input('Digite o offset:\n'))
36
37  operacao = int(input('Digite 1 para encriptar e 2 para
       decriptar a mensagem:\n'))
38
39  if operacao == 1:
40          encripte_msg(msg, offset)
41  elif operacao == 2:
42          decripte_msg(msg, offset)
43  else:
44          print('Erro, operacao invalida')
```

8.4 O código deve ser levemente alterado, deixando apenas as funções e a variável global alfabeto. Feito isso, basta salvar o arquivo em um diretório corrente, assim só poderá ser importado se estiver neste diretório, ou salvar o arquivo dentro do diretório de instalação do Python, na pasta *Lib*, assim ele poderá ser importado de qualquer lugar.

cipher_modulo

```
alfabeto = 'abcdefghijklmnopqrstuvwxyz'

def encripte_msg(mensagem, offset):
    nova_msg = str()
    arquivo = 'mensagem_encriptada.txt'

    with open(arquivo, 'w') as arquivo_encriptado:
        for i in mensagem.lower():
            if i in alfabeto:
                index = alfabeto.find(i)
                nova_msg += alfabeto[(index + offset)
                   % 26]
            else:
                nova_msg += i

        arquivo_encriptado.write(nova_msg)

    print('A mensagem foi encriptada:', nova_msg, 'E
       escrita no arquivo', arquivo, sep='\n')

def decripte_msg(mensagem, offset):
    nova_msg = str()
    arquivo = 'mensagem_decriptada.txt'

    with open(arquivo, 'w') as arquivo_decriptado:
        for i in mensagem.lower():
            if i in alfabeto:
                index = alfabeto.find(i)
                nova_msg += alfabeto[(index - offset)
                   % 26]
            else:
                nova_msg += i

        arquivo_decriptado.write(nova_msg)

    print('A mensagem foi decriptada:', nova_msg, 'E
       escrita no arquivo', arquivo, sep='\n')
```

Feito isso, em um código qualquer pode-se importar o módulo e usar suas funções.

```
>>> import cipher_modulo
>>> cipher_modulo.encripte_msg('ola', 1)
A mensagem foi encriptada:
```

```
pmb
E escrita no arquivo
mensagem_encriptada.txt
```

8.5 Python Interativo

```
>>> def multi_multiplicao(*args):
  r = 1
  for i in args:
    r *= i
  return r

>>> multi_multiplicao(2)
2
>>> multi_multiplicao(2, 2, 2, 2)
16
```

8.6 Sim, é possível. Os primeiros argumentos devem ser do tipo *args, ou seja, sem palavras chaves, enquanto que os últimos com palavras chave. Assim, o Python irá identificar qual argumento vai para args e qual para kwargs.

```
>>> def foo(*args, **kwargs):
  print(len(args))
  print(len(kwargs))


>>> foo(1, 2, k=1)
2
1
```

8.7 Não, porque, por convenção, primeiro devem ser os argumentos do tipo *args e depois os argumentos do tipo **kwargs, sem misturar a ordem.

8.8 Não é possível, pela mesma justificativa da questão anterior: primeiro devem ser os argumentos do tipo *args e depois os argumentos do tipo **kwargs, sem misturar a ordem.

```
>>> def foo(**kwargs, *args):

SyntaxError: invalid syntax
```

8.9 fib.py

```
1 def fib(n):
2         if n == 1 or n == 2:
3                 return 1
4         return fib(n-1) + fib(n-2)
```

8.10 No caso da criação de um pacote, deve-se atentar para colocá-lo dentro do diretório de instalação do Python, para que ele possa ser acessado de qualquer pasta e também se atentar para criar um arquivo __init__.py, indicando que a pasta será um pacote Python.

Orientação a Objetos: Introdução

10.1

```
>>> class fracao():
  num = 1
  den = 1
```

10.2

```
>>> class fracao():
  def __init__ (self, n, d):
    self.num = n
    self.den = d
```

10.3

```
>>> class fracao():
  def __init__ (self, n, d):
    self.num = n
    self.den = d
  def __str__ (self):
    return '%d/%d' % (self.num, self.den)


>>> X = fracao(2, 3)
>>> print(X)
2/3
```

10.4

```
>>> class fracao():
  def __init__ (self, numerador, denominador):
    self.num = numerador
    self.den = denominador
```

```
  def __str__ (self):
    return '%d/%d' % (self.num, self.den)
  def __add__ (self, other):
    return fracao(other.den*self.num +
       self.den*other.num, self.den*other.den)
  def __sub__ (self, other):
    return fracao(other.den*self.num -
       self.den*other.num, self.den*other.den)
  def __mul__ (self, other):
    return fracao(self.num*other.num,
       self.den*other.den)
  def __truediv__ (self, other):
    return fracao(self.num*other.den,
       self.den*other.num)


>>> X = fracao(2, 3); Y = fracao(5, 7)
>>> print(X+Y, X-Y, X*Y, X/Y)
29/21 -1/21 10/21 14/15
```

10.5

```
>>> class fracao():
  #Construtor
  def __init__ (self, numerador, denominador):
    self.num = numerador
    self.den = denominador

  #Impressao, e tbm conversao p/ str
  def __str__ (self):
    return '%d/%d' % (self.num, self.den)

  #Conversao
  def __int__ (self):
    return self.num//self.den
  def __float__ (self):
    return self.num/self.den
  def __bool__ (self):
    if self.num == 0:
      return False
    else:
      return True

  #Operadores
  def __add__ (self, other):
    return fracao(other.den*self.num +
       self.den*other.num, self.den*other.den)
  def __sub__ (self, other):
```

```
    return fracao(other.den*self.num -
       self.den*other.num, self.den*other.den)
  def __mul__ (self, other):
    return fracao(self.num*other.num,
       self.den*other.den)
  def __truediv__ (self, other):
    return fracao(self.num*other.den,
       self.den*other.num)


>>> X = fracao(3, 4); Z = fracao(0, 3)
>>> int(X)
0
>>> float(X)
0.75
>>> str(X)
'3/4'
>>> bool(X)
True
>>> bool(Z)
False
```

10.6

```
>>> class fracao():
        #Construtor
        def __init__ (self, numerador, denominador):
                (numerador, denominador) =
                   self.simplifica(numerador,
                   denominador)
                self.num = numerador
                self.den = denominador
      #Impressao, e tbm conversao p/ str
        def __str__ (self):
                return '%d/%d' % (self.num, self.den)

        #Conversao
        def __int__ (self):
                return self.num//self.den
        def __float__ (self):
                return self.num/self.den
        def __bool__ (self):
                if self.num == 0:
                        return False
                else:
                        return True

        #Operadores
```

```
        def __add__ (self, other):
                return fracao(other.den*self.num +
                   self.den*other.num,
                   self.den*other.den)
        def __sub__ (self, other):
                return fracao(other.den*self.num -
                   self.den*other.num,
                   self.den*other.den)
        def __mul__ (self, other):
                return fracao(self.num*other.num,
                   self.den*other.den)
        def __truediv__ (self, other):
                return fracao(self.num*other.den,
                   self.den*other.num)

        #Metodo estatico simplificador
        @staticmethod
        def simplifica(numerador, denominador):
                menor = min(numerador, denominador)
                i = 2
                while i<=menor:
                        if denominador%i == 0 and
                           numerador%i == 0:
                                numerador //= i
                                denominador //= i
                                menor =
                                   min(numerador,
                                   denominador)
                        else:
                                i += 1
                return (numerador, denominador)

>>> X = fracao(1, 6) + fracao(1, 3)
>>> print(X)
1/2
```

```
>>> class fracao():
        def __init__ (self, *argv):
                if len(argv) == 0:
                        self.num = 0
                        self.den = 1
                elif len(argv) == 1:
                        self.num = argv[0]
                        self.den = 1
                else:
                        numerador = argv[0];
```

```
                        denominador = argv[1]
                    (numerador, denominador) =
                        self.simplifica(numerador,
                        denominador)
                    self.num = numerador
                    self.den = denominador

        #Impressao, e tbm conversao p/ str
        def __str__ (self):
                return '%d/%d' % (self.num, self.den)

        #Conversao
        def __int__ (self):
                return self.num//self.den
        def __float__ (self):
                return self.num/self.den
        def __bool__ (self):
                if self.num == 0:
                        return False
                else:
                        return True

        #Operadores
        def __add__ (self, other):
                return fracao(other.den*self.num +
                    self.den*other.num,
                    self.den*other.den)
        def __sub__ (self, other):
                return fracao(other.den*self.num -
                    self.den*other.num,
                    self.den*other.den)
        def __mul__ (self, other):
                return fracao(self.num*other.num,
                    self.den*other.den)
        def __truediv__ (self, other):
                return fracao(self.num*other.den,
                    self.den*other.num)

        #Metodo estatico simplificador
        @staticmethod
        def simplifica(numerador, denominador):
                if denominador < 0:
                        denominador = - denominador
                        numerador = - numerador
                menor = min(numerador, denominador)
                i = 2
                while i<=menor:
                        if denominador%i == 0 and
```

```
                                numerador%i == 0:
                                        numerador //= i
                                        denominador //= i
                                        menor =
                                           min(numerador,
                                           denominador)
                                else:
                                        i += 1
                    return (numerador, denominador)

>>> X = fracao(-2, -2)
>>> print(X)
1/1
>>> Y = fracao(3, -2)
>>> print(Y)
-3/2
```

10.8

```
>>> class fracao():
        #Construtor
        def __init__ (self, *argv):
                if len(argv) == 0:
                        self.num = 0
                        self.den = 1
                elif len(argv) == 1:
                        self.num = argv[0]
                        self.den = 1
                else:
                        numerador = argv[0];
                           denominador = argv[1]
                        (numerador, denominador) =
                           self.simplifica(numerador,
                           denominador)
                        self.num = numerador
                        self.den = denominador

   #Impressao, e tbm conversao p/ str
        def __str__ (self):
                return '%d/%d' % (self.num, self.den)

        #Conversao
        def __int__ (self):
                return self.num//self.den
        def __float__ (self):
                return self.num/self.den
        def __bool__ (self):
```

```
                if self.num == 0:
                        return False
                else:
                        return True

        #Operadores
        def __add__ (self, other):
                try:
                        if type(other) ==
                           type(int()):
                                return
                                   fracao(self.num
                                   + other *
                                   self.den,
                                   self.den)
                        elif type(other) ==
                           type(fracao()):
                                return
                                   fracao(other.den
                                   * self.num +
                                   self.den *
                                   other.num,
                                   self.den *
                                   other.den)
                        else:
                                raise TypeError
                except TypeError as erro:
                        print('Operação não
                           suportada', erro,
                           sep='\n')
        def __sub__ (self, other):
                try:
                        if type(other) ==
                           type(int()):
                                return
                                   fracao(self.num
                                   - other *
                                   self.den,
                                   self.den)
                        elif type(other) ==
                           type(fracao()):
                                return
                                   fracao(other.den
                                   * self.num -
                                   self.den *
                                   other.num,
                                   self.den *
                                   other.den)
```

```
                        else:
                                raise TypeError
                except TypeError as erro:
                        print('Operação não
                           suportada', erro,
                           sep='\n')
        def __mul__ (self, other):
                try:
                        if type(other) ==
                           type(int()):
                                return
                                   fracao(self.num
                                   * other,
                                   self.den)
                        elif type(other) ==
                           type(fracao()):
                                return
                                   fracao(self.num
                                   * other.num,
                                   self.den *
                                   other.den)
                        else:
                                raise TypeError
                except TypeError as erro:
                        print('Operação não
                           suportada', erro,
                           sep='\n')
        def __truediv__ (self, other):
                try:
                        if type(other) ==
                           type(int()):
                                return
                                   fracao(self.num,
                                   self.den *
                                   other)
                        elif type(other) ==
                           type(fracao()):
                                return
                                   fracao(self.num
                                   * other.den,
                                   self.den *
                                   other.num)
                        else:
                                raise TypeError
                except TypeError as erro:
                        print('Operação não
                           suportada', erro,
                           sep='\n')
```

```
        #Metodo estatico simplificador
        @staticmethod
        def simplifica(numerador, denominador):
                if denominador < 0:
                        denominador = - denominador
                        numerador = - numerador
                menor = min(numerador, denominador)
                i = 2
                while i<=menor:
                        if denominador%i == 0 and
                           numerador%i == 0:
                                numerador //= i
                                denominador //= i
                                menor =
                                   min(numerador,
                                   denominador)
                        else:
                                i += 1
                return (numerador, denominador)

>>> print( fracao(2, 3) + 2 )
8/3
>>> print( fracao(2, 3) / 2 )
1/3
```

10.9

```
>>> class fracao():
        #Construtor
        def __init__ (self, *argv):
                if len(argv) == 0:
                        self.num = 0
                        self.den = 1
                elif len(argv) == 1:
                        self.num = argv[0]
                        self.den = 1
                else:
                        numerador = argv[0];
                           denominador = argv[1]
                        (numerador, denominador) =
                           self.simplifica(numerador,
                           denominador)
                        self.num = numerador
                        self.den = denominador

   #Impressao, e tbm conversao p/ str
        def __str__ (self):
```

```
                return '%d/%d' % (self.num, self.den)

        #Conversao
        def __int__ (self):
                return self.num//self.den
        def __float__ (self):
                return self.num/self.den
        def __bool__ (self):
                if self.num == 0:
                        return False
                else:
                        return True

        #Operadores
        #Aritmeticos
        def __add__ (self, other):
                try:
                        if type(other) ==
                           type(int()):
                                return
                                   fracao(self.num
                                   + other *
                                   self.den,
                                   self.den)
                        elif type(other) ==
                           type(fracao()):
                                return
                                   fracao(other.den
                                   * self.num +
                                   self.den *
                                   other.num,
                                   self.den *
                                   other.den)
                        else:
                                raise TypeError
                except TypeError as erro:
                        print('Operação não
                           suportada', erro,
                           sep='\n')
        def __sub__ (self, other):
                try:
                        if type(other) ==
                           type(int()):
                                return
                                   fracao(self.num
                                   - other *
                                   self.den,
                                   self.den)
```

```
                        elif type(other) ==
                           type(fracao()):
                                 return
                                    fracao(other.den
                                    * self.num -
                                    self.den *
                                    other.num,
                                    self.den *
                                    other.den)
                        else:
                                 raise TypeError
              except TypeError as erro:
                        print('Operação não
                           suportada', erro,
                           sep='\n')
     def __mul__ (self, other):
              try:
                        if type(other) ==
                           type(int()):
                                 return
                                    fracao(self.num
                                    * other,
                                    self.den)
                        elif type(other) ==
                           type(fracao()):
                                 return
                                    fracao(self.num
                                    * other.num,
                                    self.den *
                                    other.den)
                        else:
                                 raise TypeError
              except TypeError as erro:
                        print('Operação não
                           suportada', erro,
                           sep='\n')
     def __truediv__ (self, other):
              try:
                        if type(other) ==
                           type(int()):
                                 return
                                    fracao(self.num,
                                    self.den *
                                    other)
                        elif type(other) ==
                           type(fracao()):
                                 return
                                    fracao(self.num
```

```
                                        * other.den,
                                        self.den *
                                        other.num)
                        else:
                                raise TypeError
                except TypeError as erro:
                        print('Operação não
                           suportada', erro,
                           sep='\n')
        #Unarios
        def __abs__ (self):
                return fracao(abs(self.num),
                   abs(self.den))
        def __invert__ (self):
                return fracao(self.den, self.num)
        def __pos__ (self):
                return self
        def __neg__ (self):
                return fracao( -self.num, self.den)

        #Metodo estatico simplificador
        @staticmethod
        def simplifica(numerador, denominador):
                if denominador < 0:
                        denominador = - denominador
                        numerador = - numerador
                menor = min(numerador, denominador)
                i = 2
                while i<=menor:
                        if denominador%i == 0 and
                           numerador%i == 0:
                                numerador //= i
                                denominador //= i
                                menor =
                                   min(numerador,
                                   denominador)
                        else:
                                i += 1
                return (numerador, denominador)
```

10.10 ClasseFracao.py

```
1    #Classe Fracao
2    class fracao():
3        #Construtor
4        def __init__ (self, *argv):
```

```
5                   if len(argv) == 0:
6                           self.num = 0
7                           self.den = 1
8                   elif len(argv) == 1:
9                           self.num = argv[0]
10                          self.den = 1
11                  else:
12                          numerador = argv[0];
                               denominador = argv[1]
13                          (numerador, denominador) =
                               self.simplifica(numerador,
                               denominador)
14                          self.num = numerador
15                          self.den = denominador
16
17  #Impressao, e tbm conversao p/ str
18          def __str__ (self):
19                  return '%d/%d' % (self.num, self.den)
20
21          #Conversao
22          def __int__ (self):
23                  return self.num//self.den
24          def __float__ (self):
25                  return self.num/self.den
26          def __bool__ (self):
27                  if self.num == 0:
28                          return False
29                  else:
30                          return True
31
32          #Operadores
33          #Aritmeticos
34          def __add__ (self, other):
35                  try:
36                          if type(other) ==
                               type(int()):
37                                  return
                                       fracao(self.num
                                       + other *
                                       self.den,
                                       self.den)
38                          elif type(other) ==
                               type(fracao()):
39                                  return
                                       fracao(other.den
                                       * self.num +
                                       self.den *
                                       other.num,
```

```
                                        self.den *
                                        other.den)
40                          else:
41                                  raise TypeError
42                  except TypeError as erro:
43                          print('Operação não
                               suportada', erro,
                               sep='\n')
44          def __sub__ (self, other):
45                  try:
46                          if type(other) ==
                               type(int()):
47                                  return
                                       fracao(self.num
                                       - other *
                                       self.den,
                                       self.den)
48                          elif type(other) ==
                               type(fracao()):
49                                  return
                                       fracao(other.den
                                       * self.num -
                                       self.den *
                                       other.num,
                                       self.den *
                                       other.den)
50                          else:
51                                  raise TypeError
52                  except TypeError as erro:
53                          print('Operação não
                               suportada', erro,
                               sep='\n')
54          def __mul__ (self, other):
55                  try:
56                          if type(other) ==
                               type(int()):
57                                  return
                                       fracao(self.num
                                       * other,
                                       self.den)
58                          elif type(other) ==
                               type(fracao()):
59                                  return
                                       fracao(self.num
                                       * other.num,
                                       self.den *
                                       other.den)
60                          else:
```

```
61                                      raise TypeError
62                  except TypeError as erro:
63                          print('Operação não
                               suportada', erro,
                               sep='\n')
64          def __truediv__ (self, other):
65                  try:
66                          if type(other) ==
                               type(int()):
67                                  return
                                       fracao(self.num,
                                       self.den *
                                       other)
68                          elif type(other) ==
                               type(fracao()):
69                                  return
                                       fracao(self.num
                                       * other.den,
                                       self.den *
                                       other.num)
70                          else:
71                                  raise TypeError
72                  except TypeError as erro:
73                          print('Operação não
                               suportada', erro,
                               sep='\n')
74          #Unarios
75          def __abs__ (self):
76                  return fracao(abs(self.num),
                           abs(self.den))
77          def __invert__ (self):
78                  return fracao(self.den, self.num)
79          def __pos__ (self):
80                  return self
81          def __neg__ (self):
82                  return fracao( -self.num, self.den)
83          #Relacionais
84          def __lt__ (self, other):
85                  return self.num/self.den <
                           other.num/other.den
86          def __le__ (self, other):
87                  return self.num/self.den <=
                           other.num/other.den
88          def __eq__ (self, other):
89                  return self.num/self.den ==
                           other.num/other.den
90          def __ne__ (self, other):
91                  return self.num/self.den !=
```

```
                    other.num/other.den
92          def __gt__ (self, other):
93                  return self.num/self.den >
                       other.num/other.den
94          def __ge__ (self, other):
95                  return self.num/self.den >=
                       other.num/other.den
96
97          #Metodo estatico simplificador
98          @staticmethod
99          def simplifica(numerador, denominador):
100                 if denominador < 0:
101                         denominador = - denominador
102                         numerador = - numerador
103                 menor = min(numerador, denominador)
104                 i = 2
105                 while i<=menor:
106                         if denominador%i == 0 and
                               numerador%i == 0:
107                                 numerador //= i
108                                 denominador //= i
109                                 menor =
                                       min(numerador,
                                       denominador)
110                         else:
111                                 i += 1
112                 return (numerador, denominador)
```

Bibliografia

Conteúdo

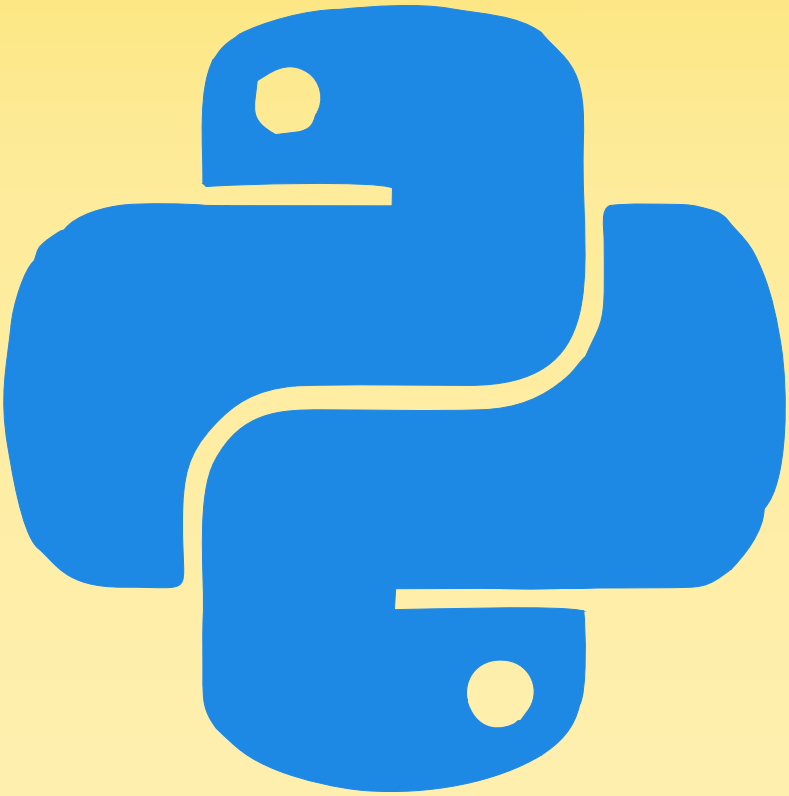
- Allen Downey, Jeffrey Elkner, Chris Meyers (2002) “How to Think Like a Computer Scientist - Learning with Python” 1ª edição.
- Camila Laranjeira (2019) “Introdução a Programação de Computadores".
- Bruno R. Preiss, Estrutura de Dados e Algoritmos com Padrões de Projetos Orientado a Objeto em Python.
- Gustavo Guanabara, Curso em Vídeo. `https://www.cursoemvideo.com/`
- Stack Overflow. `https://pt.stackoverflow.com/`
- Python Software Foundation. `https://www.python.org/`
- Wikipedia. `https://www.wikipedia.org/`
- Programiz. `https://www.programiz.com/python-programming`

- Python Tips. https://pythontips.com

- Tutorials Teacher. https://www.tutorialsteacher.com/python

Imagens

- https://www.futura-sciences.com/fonds-ecran/high-tech/informatique/
- http://hidekianagusko.com.br/free-office-wallpaper-26001-26685-hd-wallpapers/
- https://pt.wikipedia.org/wiki/Guido_van_Rossum
- https://www.imdb.com/title/tt0063929/
- https://medium.com/@nejcrodosek/how-i-speed-up-my-web-development-process-by-3
- https://br.videoblocks.com/video/animated-binary-bits-and-bytes-computer-numbe
  p5gizufznm3
- https://auditoriacidada.org.br/bibliografia-sobre-a-divida/



PETEE UFMG

UFMG