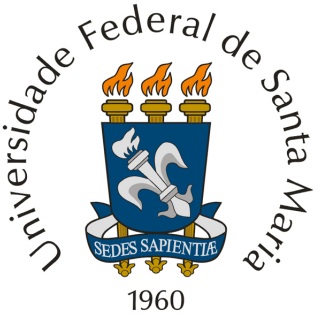


Universidade Federal de Santa Maria
Campus Cachoeira do Sul

Python 101

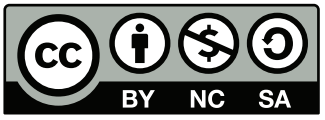
Autores:
Profª. Drª Ana Luisa Soubhia
Elias Teixeira da Costa
Flavio Luan Müller Freitas
Laís Brum Menezes
Marcos Alves dos Santos
Prof. Dr. Vinícius Maran

Esta apostila tem como objetivo apresentar um compilado de informações abordadas nos minicursos sobre *Python* ministrados pelos alunos da Universidade Federal de Santa Maria (UFSM), campus Cachoeira do Sul.

Este material é resultante da primeira etapa do projeto de ensino *Cadernos Digitais para Engenharias - Aplicações em LaTeX, Python e Jupyter*, financiado pela Universidade Federal de Santa Maria (UFSM) através do edital FIEN 2019.

Coordenador do Projeto: Prof. Dr. Vinícius Maran. Contato: vinicius.maran@ufsm.br

Sugestões sobre o material podem ser informadas nest link.



Documento criado em LaTeX

Conteúdo

1 Introdução à linguagem de Programação Python

Nesta seção são apresentados os aspectos gerais sobre a linguagem de programação Python. São abordados os tópicos relativos à história, às aplicações da linguagem e ao funcionamento de compilação de seus códigos. Ainda são apresentados tópicos referentes às IDE's mais utilizadas disponíveis, à motivação e ao porquê de se utilizar Python, às vantagens e, por fim, à sintaxe da linguagem também são abordados.

Python é uma linguagem de programação interpretada, ou seja, ao escrever um algoritmo, este não será traduzido para uma linguagem de máquina, e sim "interpretado" por outro programa, denominado interpretador.

1.1 História

Por volta dos anos 80, Van Rossum utilizava o laboratório do grupo CWI (*Distributed & Interactive Systems*) para trabalhar na criação de uma linguagem de programação chamada "ABC", juntamento com o Sistema Operacional "AMOEBA".

Nesse tempo, Rossum começou a pensar em uma nova linguagem que possuísse uma sintaxe semelhante à da "ABC" e com o acesso semelhante ao do "AMOEBA". Portanto, iniciando-se na década de 1980, e concluindo-se em 1991, foi lançada a primeira versão da linguagem "Python" 1.0.

Em relação ao porquê do nome dado à linguagem, Van Rossum seguiu o exemplo de James Gosling que deu o nome de sua linguagem (Java) devido o apreço de seus colegas por café. Isto é, segundo o site "Trytoprogram" [8], na década de 70 existia um famoso programa de comédia no canal BBC, chamado "Monty Python's Flying Circus", do qual Rossum era fã. Então, quando o projeto estava concluído, esse foi nomeado "Python" devido o gosto de seu criador pelo programa.

1.2 Aplicações

Segundo [7], Python possui inúmeros tipos de aplicações. A Figura 1 ilustra as áreas de aplicação desta linguagem.

Figura 1: *Python Applications*

Fonte: DataFlair. Disponível em:
https://data-flair.training/blogs/python-applications/

Desta forma, Python possui uma ampla gama de possibilidades de se trabalhar. A seguir são listados algumas das utilizações mais comuns em cada uma das áreas destacadas na Figura 1.

- **Desenvolvimento Web e Internet**

  Para realizar aplicações para Web utilizando Python não é um problema. A linguagem possui muitas bibliotecas para protocolos de internet como HTML, JSON e XML, por exemplo. Além disso, pode-se citar algumas bibliotecas com suas áreas de atuação:

  requests Uma biblioteca HTTP client, para realizar requests.

  beautifulsoup Uma biblioteca 'analisadora' (parser) de HTML/XML.

  feedparser Uma biblioteca 'analisadora' para feeds RSS/CDF/ATOM.

  paramiko Uma biblioteca para implementar protocolo SSH2.

  twisted Um framework para programação de redes assíncronas.

- **Aplicações a Negócios (Business)**

  Python também é muito útil e eficiente para aplicações que envolvam desenvolvimentos de ERP e de e-commerce. Como exemplo, pode-se citar:

  Tryton Uma plataforma de aplicativos de uso geral de alto nível relacionada a negócios.

  Odoo Um conjunto de códigos de negócios (open-source) Python que ajudam no gerenciamento de empresas e de organizações usando CMS, Business Intelligence engine, entre outros.

- **Aplicações GUI para Desktop**

  A maioria das interfaces gráficas criadas com Python utilizam a biblioteca padrão "tkinter". Essa biblioteca fornece a maioria dos módulos necessários para a criação de interfaces.

  Porém, também é possível utilizar outras bibliotecas e módulos de externos. Basta baixá-los e instalá-los. Na seção "Bibliotecas" será mostrado mais informações sobre como isto é feito.

- **Jogos**

  Uma das áreas com maior número de aplicações em Python é a de Games. São inúmeras as possibilidades de criação de jogos, sendo que existe um concurso semestral denominado "PyWeek".

  Dentre as bibliotecas mais utilizadas para a criação de games, destaca-se a "pygame".

- **Aplicações científicas e numéricas**

  Esta é outra área bastante volumosa no Python. Não é utopia dizer que, em muitos casos, Python é escolhido no lugar de outros softwares científicos e numéricos, como o MatLab e o SciLab, por exemplo.

- **Programação de Redes**

  Python também fornece a possibilidade de se trabalhar com programação de redes. A linguagem fornece suporte para programação de redes em "lower-level" (baixo-nível).

  Como exemplo de utilização, pode-se citar o framework **Twisted**, o mesmo utilizado para desenvolvimento web e internet.

- **Desenvolvimento de Softwares**

Para esta área, Python pode ser utilizado como uma linguagem suporte para desenvolvedores. Pode-se citar:

scons Utilizado para ocntrole de compilação.

buildbot Um framework para compilação automatizada e contínua.

roundup e trac Para gerenciamento de projetos e rastreamento de bugs.

- **Acesso a Bancos de Dados**

  Esta é uma das áreas mais fortes do Python. Existem muitos trabalhos desenvolvidos e em desenvolvimento na área de DataScience que utilizam Python.

  A linguagem fornece interfaces personalizadas para se trabalhar com MySQL, por exemplo. Para isso, pode-se citar **egenix odbc**.

  Assim como:

  durus Um banco de dados de objetos persistentes, oferecendo uma maneira fácil de usar e manter uma coleção consistente de instâncias de objetos, usadas por um ou mais processos.

  zodb Um banco de dados de objetos, muito utilizado em comunidades Pyramid e Plone e em muitas outras aplicações..

  Por fim, python também fornece API padrão para banco de dados.

- **Aplicações à Educação**

  Para apoio à educação, Python é uma ótima ferramenta, por ser simples e fácil, para os professores que pretendem usar os computadores em sala de aula. Python não se limita apenas em ensino de programação, podendo ser usada em outras atividades de ensino.

  Por estas razões, o projeto "One Laptop per Child" utilizou, em seu desenvolvimento, a linguagem Python. O ambiente Sugar, um software multiplataforma e aberto a todas as linguagens de programação, tem como uma de suas bases principais a linguagem Python.

- **Outras aplicações**

  As aplicações listadas acima são as mais comuns. Contudo, Python é utilizado para inúmeras outras aplicações, como por exemplo, em Inteligência Artificial, Robótica e Machine Learning.

1.3 Compilação

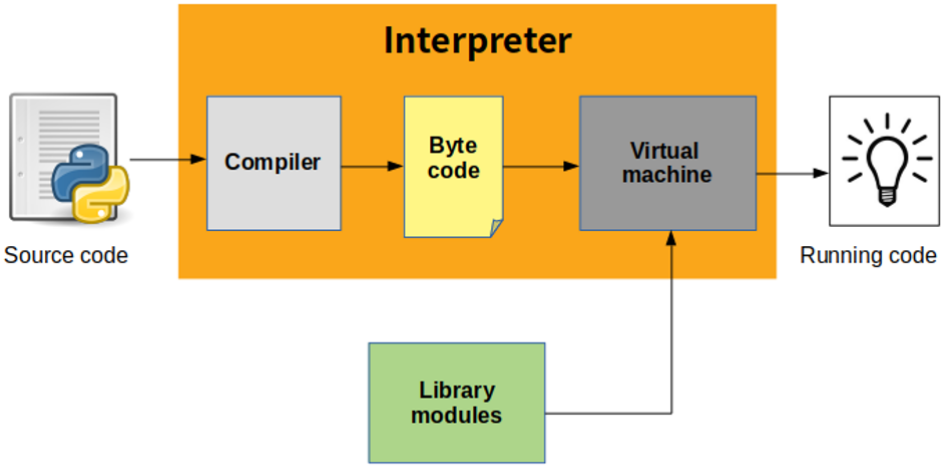
De acordo com [1], a compilação/interpretação do código em Python (.py) segue o seguinte caminho: Primeiramente o código é traduzido para *bytecode*, que é um formato binário com instruções para o interpretador. Esse *bytecode* é multiplataforma.

A seguir, o interpretador compila o código e armazena o *bytecode* em disco. Nesta etapa, quando um módulo é utilizado, o interpretador analisa o código, converte para os símbolos padrões, compila e executa na máquina virtual do Python (*Virtual Machine*).

Os arquivos ".pyc" e ".pyo" são os arquivos onde o *bytecode* é armazenado, sendo esses normal (.pyc) e otimizado (.pyo).

Ainda segundo [1], o *bytecode* pode ser empacotado junto com o interpretador em um executável, com o intuito de facilitar a distribuição do programa, sem a necessidade de se instalar o interpretador em cada máquina. A Figura 2 exemplifica este processo.

Figura 2: Compilação/Interpretação em Python



Fonte: Indian Pythonista. Disponível em:
https://indianpythonista.wordpress.com/2018/01/04/how-python-runs/

1.4 *Integrated development environment* (IDE)

De acordo com [6], existem inúmeras IDE's para se programar em Python. Dentre essa ampla gama de possibilidades, pode-se separar entre as gratuitas e as não-gratuitas.

- **Gratuitas**

  PyScripter

  Suporta debugging, auto-completion, navegação no código entre outros recursos.

- **Pagas**

  PyCharm

  Possui um conjunto de ferramentas úteis para um desenvolvimento produtivo. Além disso, o IDE fornece capacidades de alta classe para o desenvolvimento Web profissional. Possui suporte a diversos sistemas de controle de versão, integração com Github e atraves de plugin, com o Heroku. Possui gerador de Diagramas de Classe e ORM. Suporte para interpretador Python remoto. Criação de gerencia de ambientes (virtualenv).

  Wing

  Possui editor inteligente, 'powefull' debugger, fácil navegação, desenvolvimento remoto, é customizável e extensível, entre outras funcionalidades.

1.5 Por que Python?

Segundo [2], Python é uma linguagem de programação de uso geral. É, também, uma linguagem fácil de aprender, explicando porque se tornou a língua mais ensinada nas universidades.

Os interpretadores de Python estão disponíveis para os principais sistemas operacionais (Linux, Mac OS, Windows, Android, iOS, BSD, etc.).

Como descrito no início deste capítulo, Python possui aplicações em muitas áreas, e isto fornece à linguagem um maior destaque sobre outras linguagens de programação usadas na indústria. Conforme o site "Medium"[5], algumas de suas vantagens são:

- Bibliotecas de suporte extensivas;

- Recurso de integração;
- Produtividade melhorada do programador;
- Produtividade;
- Limitações ou desvantagens do Python;
- Dificuldade em usar outros idiomas;
- Fraco em computação móvel;
- Fica lento em velocidade;
- Erros de tempo de execução;
- Camadas de acesso ao banco de dados subdesenvolvidas.

1.6 Sintaxe da linguagem

Python possui um conjunto de regras de escrita que definem como um algoritmo é interpretado, possuindo um layout visual relativamente organizado e utilizando, com frequência, palavras em Inglês.

Python usa indentação como delimitação de blocos. A seguir, podemos ver dois exemplos de indentação, apresentando dois modos.

Modo errado de indentação:

```
def f():
x = 42
return x

print(f())
```

Modo correto de indentação:

```
def f():
    x = 42
    return x

print(f())
```

Se o bloco tem apenas um comando, pode-se escrever tudo em uma linha:

```
if resposta == 42: print('É a resposta para tudo!')
```

Para colocar comentários no código, utiliza-se *hashtag* (comentários de uma linha), aspas simples e aspas duplas (comentários de várias linhas):

```
# Exemplo de comentário em uma linha!

'''
Exemplo de
comentário utilizando
aspas simples.
'''

"""
Exemplo de
comentário utilizando
aspas duplas.
"""
```

Ao decorrer da apostila, serão apresentadas outras regras de sintaxe, de acordo com os conteúdos de cada seção da apostila.

2 Download e Instalação de Ferramentas para Desenvolvimento

A forma mais comum para programar em Python é através do uso de um ambiente de programação, também conhecido como ambiente integral de desenvolvimento (*Integrated Development Environment – IDE)*. Além disso, é necessário que seja instalado na máquina o interpretador do Python na versão a qual se deseja programar.

Diante disso, neste capítulo é apresentado o passo a passo de como baixar e instalar tanto o interpretador Python quanto a IDE Pycharm. O passo a passo é apresentado para o sistema operacional Windows[1][2].

2.1 Download e Instalação do Interpretador Python

Para se realizar esta tarefa, basta acessar o link no site do Python, na seção de downloads. O link é este: https://www.python.org/downloads/.

Ao acessar este link, é possível visualizar uma página semelhante à Figura 3:

Dessa forma, para baixar a versão mais atual do interpretador do Python, basta clicar em Download (Figura 4).

Após essa etapa, o download de um executável será iniciado. Ao fim do download, basta executar o arquivo com um clique-duplo.

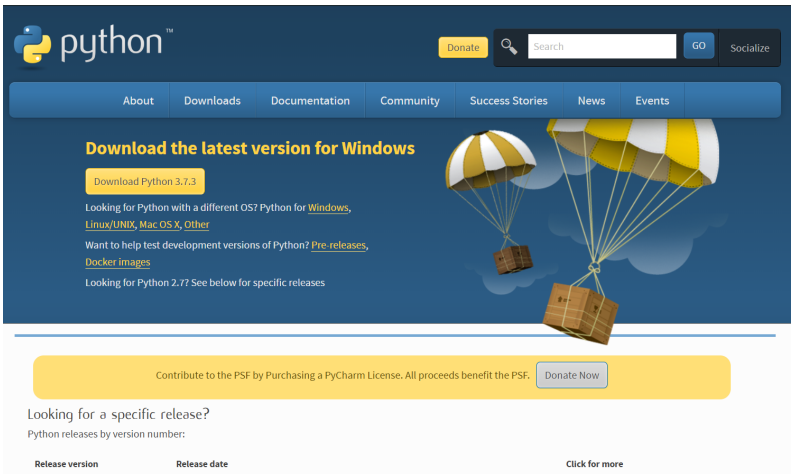
Caso haja interesse ou necessidade, é possível efetuar o download e a instalação de outras versões disponível no mesmo site. Essas estão na mesma página de download, como é possível observar na Figura 5:

O arquivo executável denominado "python-3.7.4.exe" apresenta a primeira janela exibida a seguir. A instalação recomendável segue-se clicando em *Install Now*, onde será instalado em um diretório padrão (Figura 6).

Uma tela de progresso irá aparecer (Ver Figura 7). Após a finalização do mesmo, basta clicar em *Close* (Ver Figura 8) para concluir a instalação.

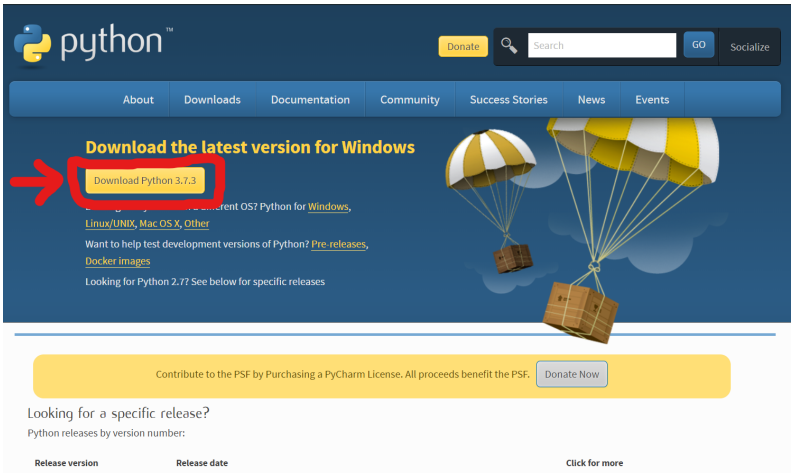
[1]Instruções para Linux: https://python.org.br/instalacao-linux/
[2]Instruções para Mac OS X: https://python.org.br/instalacao-mac/

Figura 3: Site Python Downloads



Fonte: Python. Disponível em: https://www.python.org/downloads/

Figura 4: Site Python Downloads



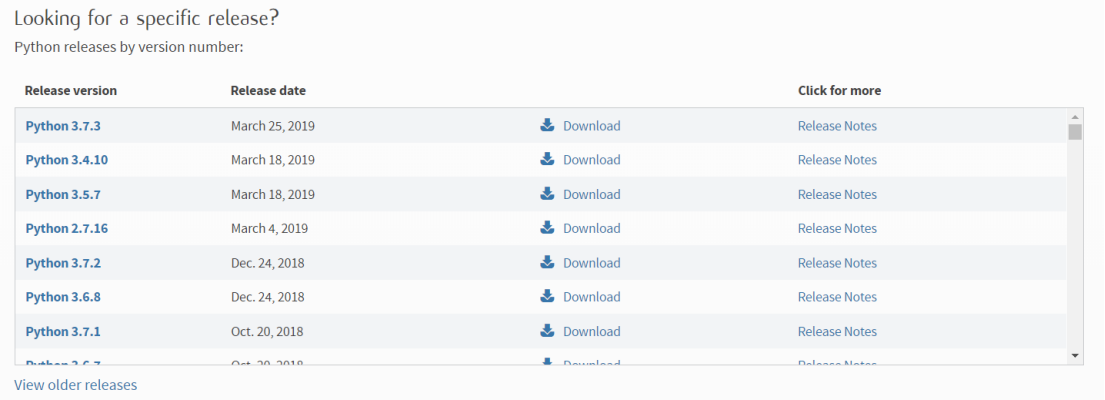
Fonte: Python. Disponível em: https://www.python.org/downloads/

2.2 Baixando e instalando a IDE PyCharm

Para instalar o PyCharm, basta acessar o site da empresa *JeBrains*. A seguir, na aba *Tools* e na seção *IDEs*, basta selecionar a opção *PyCharm*. A página será atualizada e, então, basta clicar em *Download Now*.

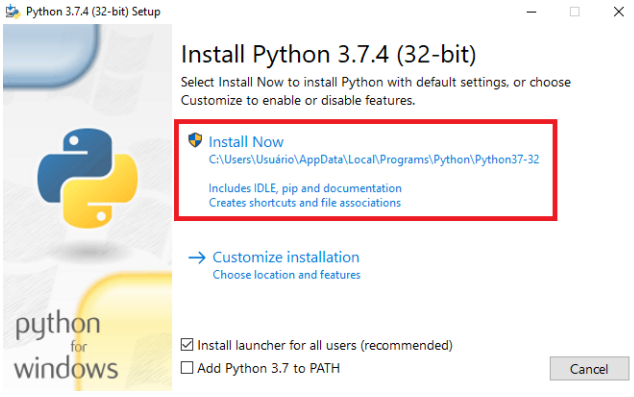
Por fim, na nova página atualizada, basta selecionar a opção *Download* na

Figura 5: Site Python Downloads



Fonte: Python. Disponível em: https://www.python.org/downloads/

Figura 6: Janela de instalação do Python



Fonte: Autores

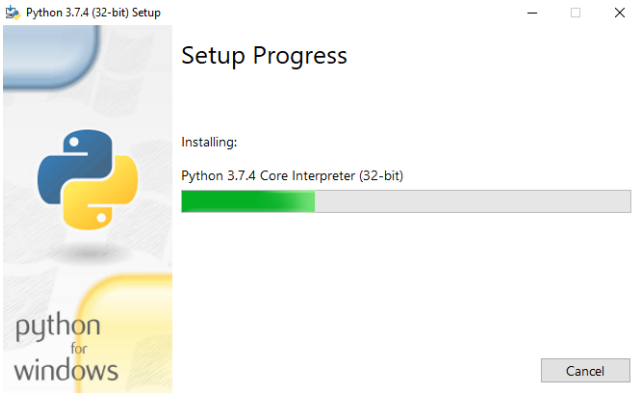
área da versão *Community*. O link direto para esta página é este: https://www.jetbrains.com/pycharm/download/#section=windows, e a Figura 9 indica a opção *Community* que deve ser selecionada.

Ressalta-se que a versão *Community* foi selecionada por ser a opção gratuita deste IDE. A seguir, será realizado o download de um executável. Assim, basta executar este instalador e aceitar as permissões de segurança.

Dessa forma, uma janela como a representada na Figura 10 será aberta.

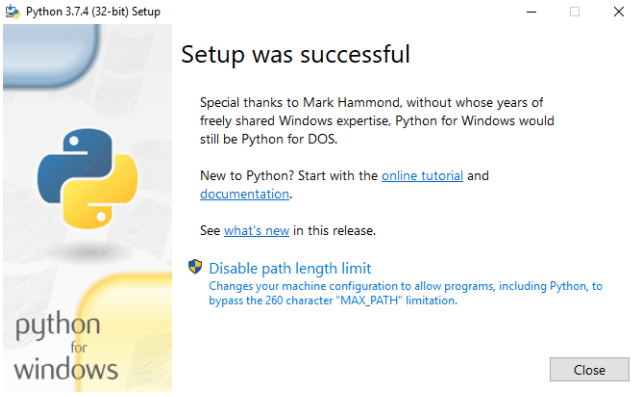
Com isso, para concluir a instalação, basta ir clicando em "*Next* >", conforme as Figuras 11, 12, 13 e 14, indicadas abaixo.

Figura 7: Janela de progresso do instalador



Fonte: Autores

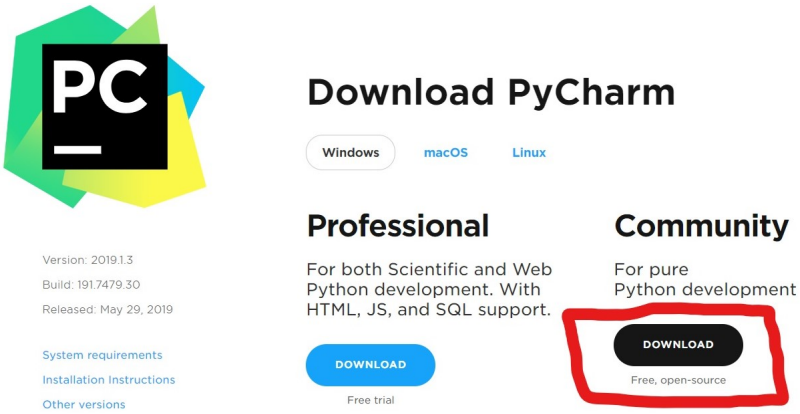
Figura 8: Janela de conclusão da instalação



Fonte: Autores

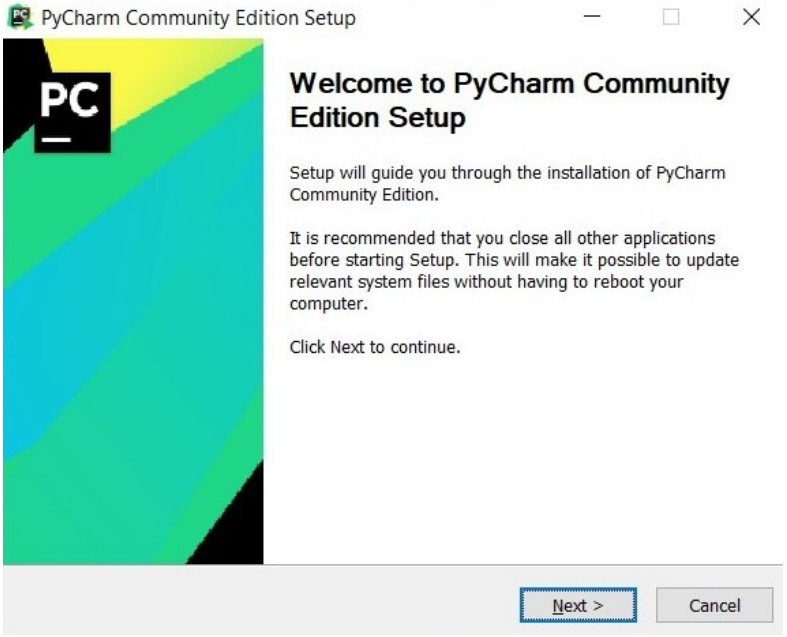
Após esses passos, basta selecionar se deseja reiniciar agora (*Reboot now*) ou reiniciar depois manualmente (*I want to manualy reboot later*) e clicar em "*Finish*". O PyCharm estará instalado e pronto para uso.

Figura 9: Site para download do PyCharm



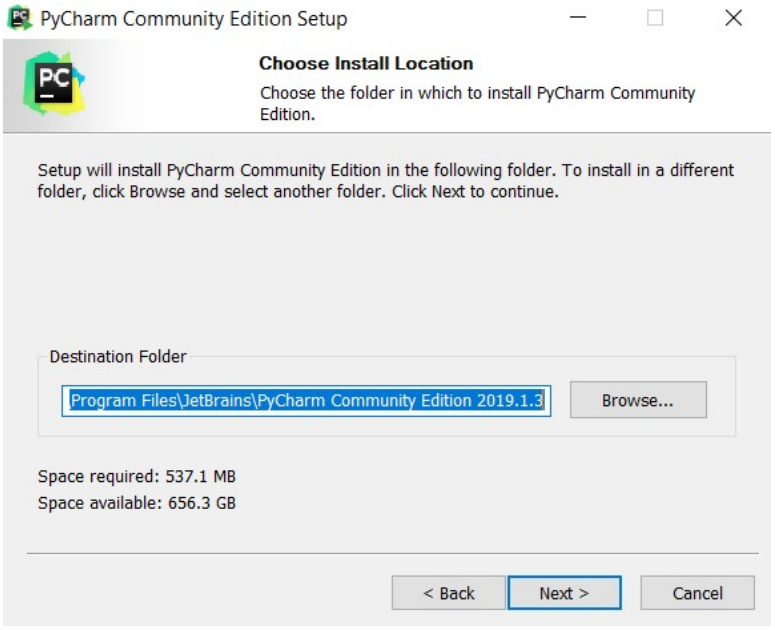
Fonte: Autores

Figura 10: Instalando o PyCharm



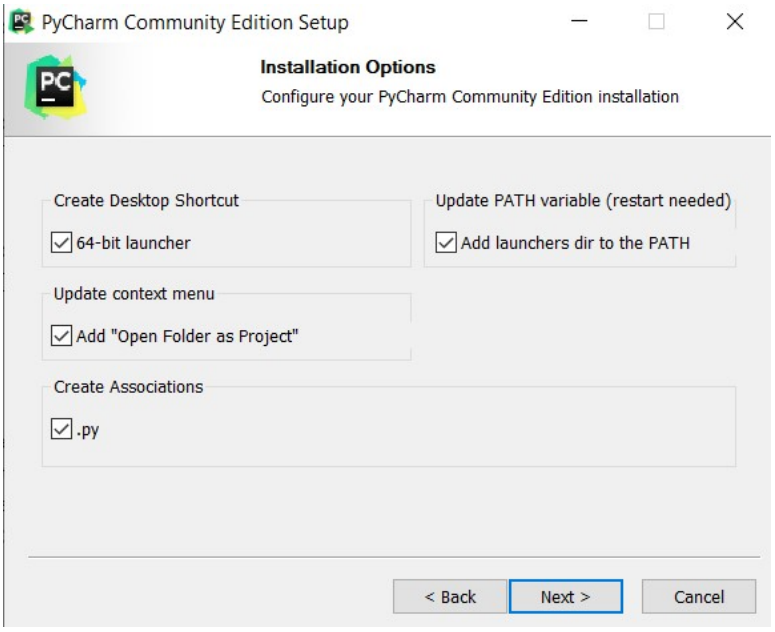
Fonte: Autores

Figura 11: Selecionando o local de instalação



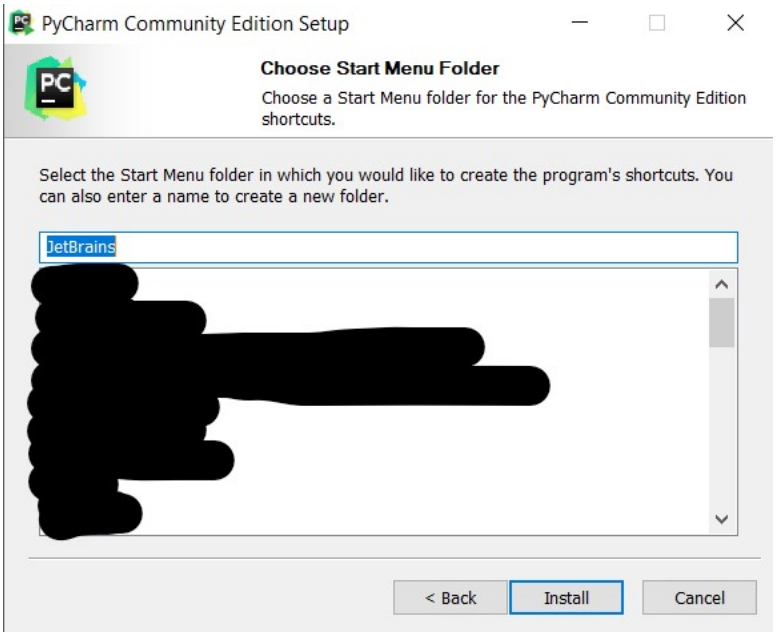
Fonte: Autores

Figura 12: Configurações Opcionais de instalação



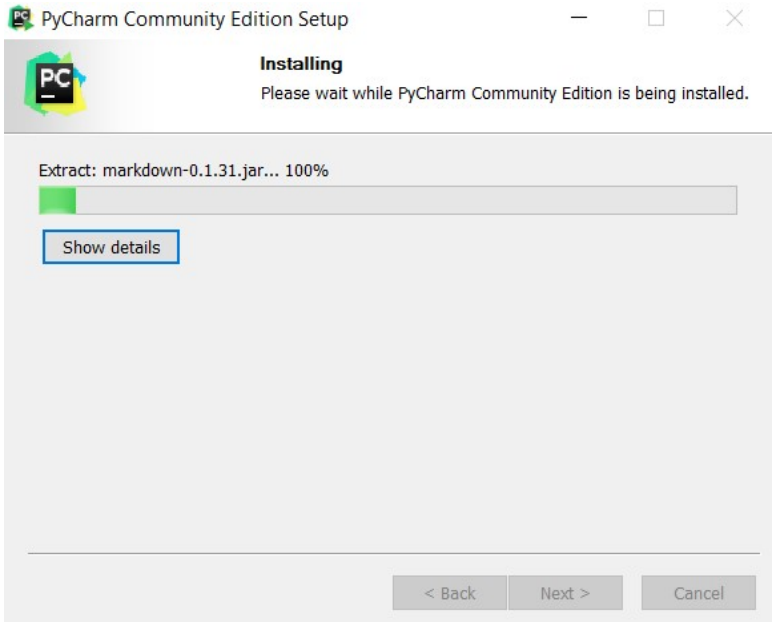
Fonte: Autores

Figura 13: Selecionando a Pasta de Menu Iniciar



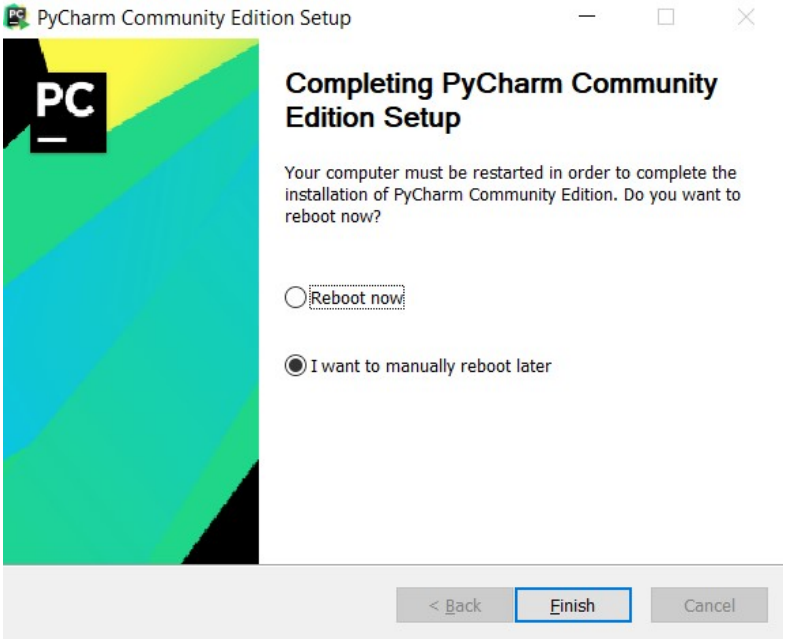
Fonte: Autores

Figura 14: Instalação em andamento



Fonte: Autores

Figura 15: Terminando a instalação



Fonte: Autores

3 Conceitos Básicos

O Python é uma linguagem de programação de alto nível e interpretada. Portanto, para que se possa programar em Python, é necessário a utilização (junto à IDE escolhida) do interpretador do Python. Para este material, foi utilizado o interpretador Python 3.7.2 e como IDE utilizou-se o Pycharm Community Edition 2019.1.1.

Nesta unidade serão abordados os seguintes itens: Hello World, Variáveis, Expressões aritméticas, Expressões lógicas, Entrada e saída de dados, Atribuição de variáveis e Bibliotecas.

3.1 *Hello World*

Como de costume na programação, a impressão da frase ***Hello World*** é utilizada como o primeiro programa ao se iniciar numa nova linguagem. Isso é apenas um costume adotado por alguns programadores, não sendo regra.

Contudo, para o fim a que se destina este material, essa impressão pode ser utilizada para se fazer alguns apontamentos introdutórios sobre a linguagem Python.

Portanto, abaixo é representada a forma básica de como imprimir a frase ***Hello World*** em Python.

```
print('Hello World!')
```

Como é possível observar, para a impressão da frase em questão, basta uma única linha de código. Contudo, para que isso seja possível, é necessário o uso da função ***print()***, a qual vem como padrão no interpretador do Python. Com essa função, o programador pode apenas digitar qualquer caractere possível dentro do abre e fecha aspas, sendo que essas aspas podem ser tanto aspas simples (‘’) como aspas duplas (“”).

Segundo [3], em Python 3 a instrução ***print()*** é uma função devido o fato do uso dos parênteses. Contudo, em Python 2, essa instrução não é uma função pois não faz uso dos parênteses, como no exemplo abaixo:

```
>>> print 'Hello World!'
```

Ambas formas são parecidas, porém existe distinção entre uma função e uma instrução. Nos próximos tópicos isso fará mais sentido.

3.2 Variáveis

Variáveis podem ser entendidas como um dos assuntos mais básicos presentes em todas as linguagens de programação. É necessário que a linguagem consiga entender qual o tipo de valor que o programador está utilizando, pois é dessa forma que ela pode executar o programa de forma correta.

Em Python, pode-se descobrir qual o tipo da variável através do interpretador utilizado. Para isso, basta utilizar a função ***type()*** e a variável dentro do parênteses desta função no prompt de comando do interpretador ou na IDE utilizada, como segue no exemplo mostrado abaixo:

```
>>> type(1)
<class 'int'>
>>>
>>> type(1.1)
<class 'float'>
>>>
>>> type(1 + 1.1j)
<class 'complex'>
>>>
>>> type('Python')
<class 'str'>
>>>
>>> type([1, 1.1, 1 + 1.1j, 'Python'])
<class 'list'>
>>>
```

Como é possível observar, o número ***1*** corresponde ao tipo ***int***, que significa uma variável inteira. O número ***1.1*** corresponde ao tipo ***float***, que significa uma variável com ponto flutuante. O número ***1 + 1.1j*** corresponde ao tipo ***complex***, que representa uma variável complexa. A frase ***Minicurso*** corresponde ao tipo ***str***, que significa uma variável do tipo *string*. O último exemplo da utilização da função ***type()*** resultou num tipo ***list***, que é uma lista (este assunto é abordado nos próximos tópicos).

Para a declaração de uma variável, basta fazer sua atribuição. Devido ao alto nível da linguagem Python, torna-se desnecessário se declarar uma variável do tipo

int, ***float*** ou ***str***, como se faz em outras linguagens de programação, como C, por exemplo.

Porém, existem algumas restrições para nomes de variáveis. Para variáveis, não se pode iniciar seu nome com um número e não se pode utilizar caracteres ilegais. Além disso, não se podem utilizar palavras-chave do Python como variáveis. Na Figura 16 são listadas algumas palavras-chave do Python 3, segundo [3].

Figura 16: Palavras-chave em Python 3

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

Fonte: Downey(2015) [3].

Segundo [1], as variáveis são criadas aos serem atribuídas e são destruídas pelo coletor de lixo (*garbage colector*), quando o interpretador entende que não há mais referências a ela.

Os tipos de variáveis podem ser classificados em mutáveis ou imutáveis. Os tipos mutáveis permitem que as variáveis atribuídas a eles sejam alteradas, ao passo que as variáveis dos tipos imutáveis não podem ser alteradas após sua atribuição.

Dessa forma, abaixo é demonstrado como se declarar alguns tipos de variáveis. Destaca-se que, para variáveis do tipo mutáveis, sua alteração para outro tipo de variável é feita de forma dinâmica, como no exemplo abaixo em que a variável ***d*** é um inteiro, depois é um número complexo e por fim se torna uma lista.

```
>>> d = 7
>>> print(d)
7
>>>
>>> d = 7 + 5j
>>> print(d)
(7+5j)
```

```
>>>
>>> d = [7, 7 + 5j, 'variavel dinamica!']
>>> print(d)
[7, 7 + 5j, 'variavel dinamica!']
>>>
```

Por fim, vale destacar que existem variáveis que podem assumir o tipo ***bool***, representando valores booleanos. Ou seja, são variáveis que podem assumir o resultado ***True*** quando verdadeiro, ou ***False*** quando falso.

3.3 Expressões aritméticas

Outro assunto de grande importância ao universo da programação são as expressões utilizadas em cada linguagem. Em Python, as expressões aritméticas não fogem ao comum encontrado em outras linguagens de programação e são de fácil entendimento.

Os operadores *+*, *-*, ***, */* e *%* executam, respectivamente, adição, subtração, multiplicação, divisão de dois números e o último retorna o valor do resto da divisão entre dois números.

Desses operadores, estendem-se outras duas operações, que são, respectivamente, a operação de exponenciação representada por ****, e a operação de divisão que resulta na parte inteira da divisão representada por *//*. Destaca-se que a operação de divisão simples */* sempre resulta num valor do tipo ***float***. Abaixo estão representadas essas operações [3].

```
>>> 11 + 7
18
>>> 18 - 11
7
>>> 7 * 11
77
>>> 77 / 11
7.0
>>> 7 % 11
7
>>> 7 ** 11
1977326743
```

[3] A linguagem Python suporta sobrecarga de operadores para outros tipos de dados. Mais informações podem ser encontradas em: https://docs.python.org/3/reference/datamodel.html#specialnames

```
>>> 1977326743 // 11
179756976
>>> 1977326743 // 11
179756976.636363
```

3.4 Expressões lógicas

Seguindo o raciocínio das expressões aritméticas, as expressões lógicas possuem demasiada relevância para a lógica da programação. Isso devido o fato de serem utilizadas, em muitos casos, em maior número do que as expressões aritméticas. Seja em laços de repetição ou em comparações lógicas, as expressões lógicas, os operadores relacionais e os operadores bitwise são muito relevantes.

Em Python, as expressões lógicas, os operadores relacionais e os operadores bitwise, são apresentados na Tabela 1.

Diante disso, destaca-se que em Python é possível a criação e a análise de intervalos. Ou seja, é possível utilizar um intervalo como uma estrutura lógica. Um exemplo dessa utilização é mostrado na abaixo.

```
>>> a = 5
>>> if 1 <= a <= 6: print('A variável a está contida no intervalo de 1
                                    até 6!')
...
A variável a está contida no intervalo de 1 até 6!
```

3.5 Entrada e Saída de Dados

Em Python, a forma básica e geral de entrada de dados é através da função **"input()"**. Essa é a função que permite ao usuário do programa inserir um dado, sendo que o mesmo pode ser endereçado a uma variável previamente programada.

Essa função **"input()"**, converte tudo o que for inserido em uma variável do tipo string. Ou seja, para que se possa inserir números (int, float ou complex), é necessário que sejam feitas as devidas conversões. Para isso, basta utilizar a função do tipo de variável que se deseja converter com a função **"input()"** dentro dos parênteses.

Operadores	**Descrição**
and	Retorna um valor verdadeiro se, e somente se, receber duas expressões verdadeiras.
or	Retorna um valor falso se, e somente se, receber duas expressões falsas.
not	Retorna verdadeiro se receber uma expressão falsa, e vice-versa.
is	Retorna verdadeiro se, e somente se, receber duas expressões cujos valores são iguais.
in	Retorna verdadeiro se, e somente se, receber um valor contido numa lista, tupla, num dicionário, etc.
X « Y	Retorna X com os bits descolados à esquerda por Y lugares.
X » Y	Retorna X com os bits deslocados à direita por Y lugares.
X	Retorna o complemento de X. É equivalente a: - X - 1.
X ^. Y	É um bitwise exclusivo (ou XOR a cada bit). O bit da saída é o mesmo que o da entrada (X) se o bit de Y for 0, e é o complemento do bit de entrada (X) se esse bit em Y for 1.
X == Y	Retorna verdadeiro se, e somente se, X for igual a Y.
X != Y	Retorna verdadeiro se, e somente se, X for diferente de Y.
X > Y	Retorna verdadeiro se, e somente se, X for maior que Y.
X < Y	Retorna verdadeiro se, e somente se, X for menor que Y.
=> ou =>	Retorna verdadeiro se, e somente se, X for maior ou igual a Y.
<= ou <=	Retorna verdadeiro se, e somente se, X for menor ou igual a Y.

Tabela 1: Operadores lógicos, relacionais e bitwise. Fonte: Autores.

Para converter um número inserido pelo usuário em um número inteiro, basta escrever ***tipo_para_o_qual_se_deseja_converter(input())***. Por exemplo, para converter a leitura de um número para inteiro, utiliza-se ***int(input())***, se a conversão deve ser feita para um número com ponto flutuante, basta escrever ***float(input())***.

Além disso, a função ***input()*** permite ao programador escrever alguma mensagem a ser lida pelo usuário no momento da inserção do dado. Para isso, basta escrever a mensagem dentro dos parênteses da função ***input()***, estando contida dentro das aspas (simples ou normal).

O exemplo apresentado abaixo ilustra os casos mencionados anteriormente.

```
>>> a = input('\Digite um número: ')
Digite um número: 7
>>> a
```

```
'7'
>>> # O número 7 é representado por um tipo string. Por isso está
                                        dentro das aspas.
...
>>> # Para que esse número seja entendido como um tipo numérico, é
                                        necessário convertê-lo, como segue:
...
>>> # Conversão de '7' para inteiro:
...
>>> a = int(a)
>>> a
7
>>>
>>> # Conversão de '7' para float:
...
>>> a = float(a)
>>>
>>> a
7.0
>>>
>>> # Conversão de '7' para complexo:
...
>>> a = complex(a)
>>> a
(7+0j)
>>>
```

Para a conversão de variáveis entre os tipos possíveis, basta seguir o mesmo raciocínio acima. Isso vale tanto para conversão de um tipo ***str*** para um tipo ***int***, ***float*** ou ***complex***, como para a conversão de um tipo ***float***, ***int*** ou ***complex*** para um tipo ***str***.

Para a saída de dados em Python, a forma mais básica e geral é através do uso da função ***print()***. Essa função permite ao programador imprimir na tela do usuário as informações desejadas.

O funcionamento dessa função é simples. Para se imprimir um texto, basta colocá-lo dentro de aspas (simples ou normal). Para se imprimir os valores de variáveis, basta escrever o nome da variável desejada. Caso a variável for diferente do tipo *string*, deve-se acrescentar uma vírgula para separá-la de outras variáveis ou do texto dentro das aspas; caso a variável for do tipo *string*, então deve-se acrescentar o operador *+*, que simbolizará a concatenação de *strings*. Abaixo é apresentado um exemplo de utilização da função ***print()***.

```
>>> a = 7
>>> b = 5
>>> texto = 'Apostila Python'
```

```
>>> print('\nAs variáveis a, b e texto são: ', a, b, texto)

As variáveis a, b e texto são: 7 5 Apostila Python

>>> A variável texto é: Apostila Python
>>>
```

Caso haja interesse e/ou necessidade, os valores (quaisquer) das variáveis a serem impressas através da função ***print()*** podem ser convertidos ao tipo *string* e concatenados com o operador ***+***.

A concatenação de *strings* é, de forma prática, a soma de uma *string* com outra. Por exemplo, ***a = 'Um'***, ***b = ' Dois'*** e ***c = a + b***. Se a variável ***c*** for printada, o resultado será ***"Um Dois"***. Dessa forma, qualquer *string* também pode ser repetida utilizando o comando ***n * string***, onde ***n*** é o número (inteiro) de repetições e ***string*** é a *string* a ser repetida. Abaixo é exemplificado este processo:

```
>>> s = 'Apostila de Python'
>>> d = ' é top!'
>>> f = s + 5*d
>>> f
'Apostila de Python é top! é top! é top! é top! é top!'
>>>
>>> f = s + 100*d
>>> f
'Apostila de Python é top! é top! é top! é top! é top! é top! é top! é
                                    top! é top! é top! é top! é top! é
                                    top! é top! é top! é top! é top! é
                                    top! é top! é top! é top! é top! é
                                    top! é top! é top! é top! é top! é
                                    top! é top! é top! é top! é top! é
                                    top! é top! é top! é top! é top! é
                                    top! é top! é top! é top! é top! é
                                    top! é top! é top! é top! é top! é
                                    top! é top! é top! é top! é top! é
                                    top! é top! é top! é top! é top! é
                                    top! é top! é top! é top! é top! é
                                    top! é top! é top! é top! é top! é
                                    top! é top! é top! é top! é top! é
                                    top! é top! é top! é top! é top! é
                                    top! é top! é top! é top! é top! é
                                    top! é top! é top! é top! é top! é
                                    top! é top! é top! é top! é top! é
                                    top! é top! é top! é top! é top! é
                                    top! é top! é top!'
>>>
```

Uma outra forma de se utilizar a função ***print()***, é através da interpolação das variáveis dentro do espaço destinado ao texto (dentro das aspas). Para isso, deve-se saber previamente qual o tipo de variável e escrever a função com o operador **%** junto da letra que corresponde à variável a ser interpolada. Na Tabela 2 estão indicados os símbolos usados na interpolação das variáveis.

Símbolo	**Tipo de Variável**
%s	String
%d	Inteiro
%f	Float
%o	Octal
%x	Hexadecimal
%e	Real exponencial
%%	Sinal de porcentagem

Tabela 2: Símbolos utilizados para interpolação de strings. Fonte: Autores.

Abaixo é exemplificado o uso desses símbolos na interpolação de strings com a função ***print()***.

```
>>> a = 7
>>> b = 5
>>> print("\nHORÁRIO: %02dh%02d" %(a,b))
HORÁRIO: 07h05
>>>
>>> print("\nPORCENTAGEM: %.0f.%.0f%%" %(a,b))
PORCENTAGEM: 7.5%
>>>
>>> print("\nExponencial: %.3e" %a)
Exponencial: 7.000e+00
>>>
>>> print("\nHexadecimal: %x, Decimal: %d, Octal: %o " %(10,10,10))
Hexadecimal: a, Decimal: 10, Octal: 12
>>>
```

3.6 Atribuição

Em Python, assim como em uma ampla gama de linguagens de programação, as atribuições são feitas através do operador =, sendo que o valor à esquerda do operador recebe o valor à direita desse. Dessa forma, as variáveis são atribuídas, as listas, vetores e outras estruturas são iniciadas.

Um exemplo simples é mostrado abaixo.

```
>>> # Atribuição de variáveis:
...
>>> a = 7
>>> b = 7 + 7.7j
>>> c = 'Texto....'
>>>
>>> # Atribuição de listas, Tuplas e Dicionários
...
>>> lista = []
>>> tupla = ()
>>> dicionario = {}
>>>
```

Também é possível fazer novas atribuições à mesma variável, fazendo com que a mesma assuma valores de tipos diferentes em momentos diferentes do programa. Isso é exemplificado abaixo.

```
>>> a = 7
>>> print('a = ', a, '\nTipo: ', type(a))
a =  7
Tipo:  <class 'int'>
>>>
>>> a = a + 7j
>>> print('a = ', a, '\nTipo: ', type(a))
a =  (7+7j)
Tipo:  <class 'complex'>
>>>
>>> a = 'Muito simples, não?!'
>>> print('a = ', a, '\nTipo: ', type(a))
a =  Muito simples, não?!
Tipo:  <class 'str'>
>>>
```

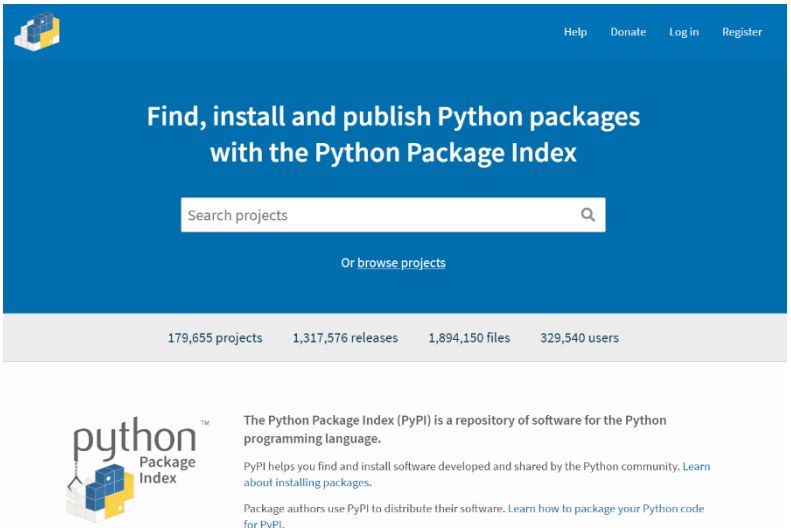
3.7 Bibliotecas

Uma das grandes vantagens de se programar em Python é o grande número de bibliotecas disponíveis para inúmeras operações diferentes.

Muitas bibliotecas vêm instaladas no interpretador do Python, porém caso haja interesse, é possível baixar outras bibliotecas que disponibilizam outras funções, classes, entre outros.

Para isso, existe o repositório oficial do Python chamado PyPi (*Python Package Index*). Nele, é possível buscar e encontrar um grande acervo de bibliotecas (tanto para Python como para outras linguagens). A Figura 17 mostra o site.

Figura 17: Site do repositório PyPi



Fonte: Autores. Disponível em: https://pypi.org/

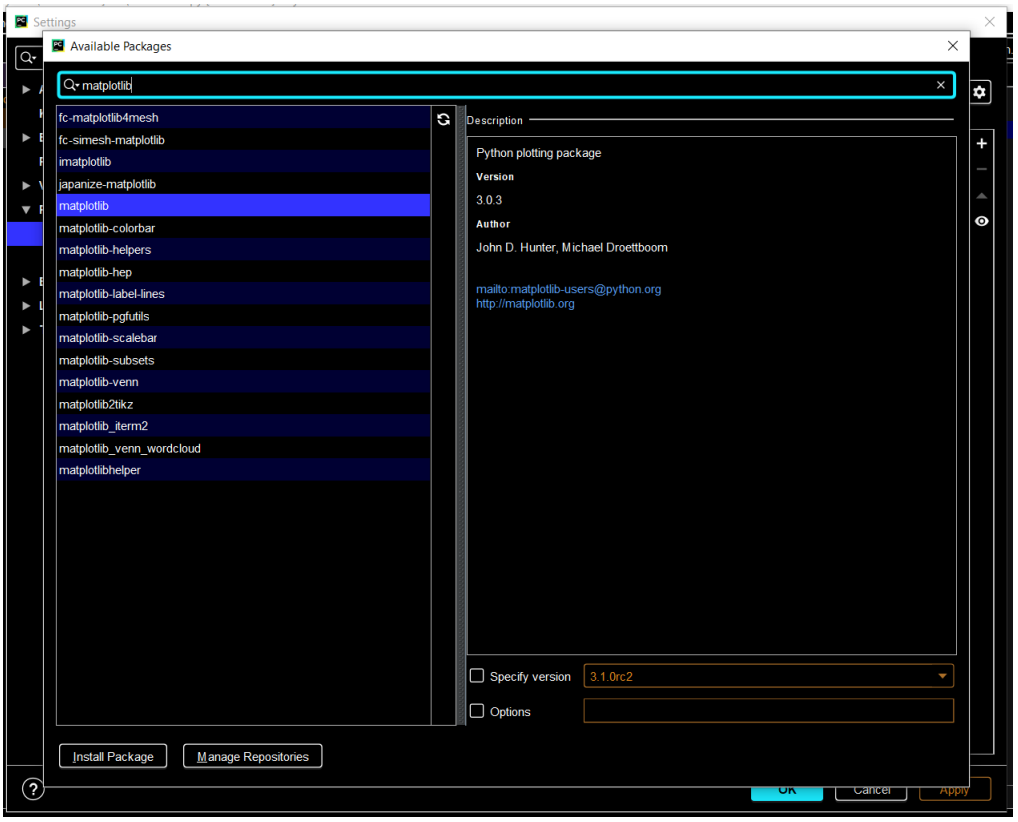
Após encontrar uma biblioteca do interesse do usuário, é possível realizar sua instalação de, pelo menos, duas formas.

A primeira forma é através da IDE utilizada. Neste caso, utilizando o PyCharm, basta seguir o caminho: *File* → *Settings* → *Project* → *Project Interpreter* → *Install (ícone de "+" no canto superior direito da tela)*. Este processo pode ser abreviado apertando *"Ctrl + Alt + S"* → *Project* → *Project Interpreter* → *"Alt + Insert"*. Nesta etapa, basta digitar o nome da biblioteca encontrada no PyPi e seguir sua instalação de forma intuitiva. A Figura 18 apresenta um exemplo de instalação de biblioteca utilizando o PyCharm.

Outra forma de se instalar uma biblioteca é através do prompt de comandos do sistema operacional (ou no próprio prompt do PyCharm). Para isso, basta escrever: ***pip install <nome_da_biblioteca>***. Na Figura 19 consta um exemplo de instalação através do prompt do Windows.

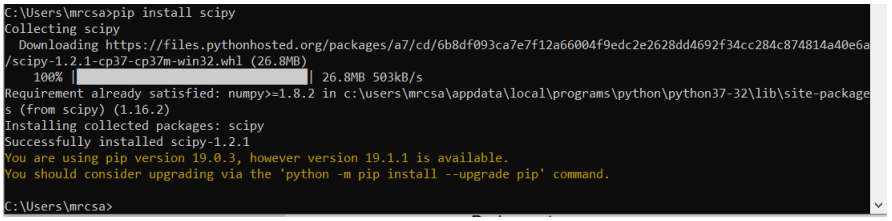
Para se utilizar a biblioteca, basta fazer uso do comando ***import*** no início do programa, seguido do nome da biblioteca. Além disso, pode-se importar apenas uma classe ou função de uma biblioteca. Para isso, escreve-se ***from <nome_da_biblioteca> import <nome_do_objeto_importado>***.

Figura 18: Instalação da biblioteca *matplotlib* através da IDE PyCharm



Fonte: Autores.

Figura 19: Instalação da biblioteca *scipy* através do prompt de comandos do Windows

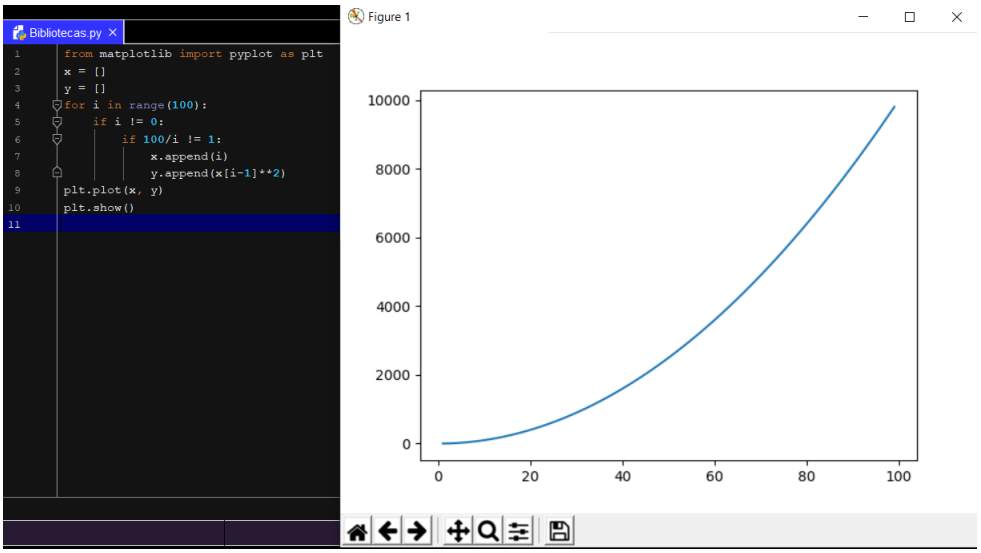


Fonte: Autores.

Esses comandos são muito úteis aos se trabalhar com muitas classes e métodos na programação orientada a objetos. Abaixo consta um exemplo de utilização do comando ***import***.

```
from math import *
print('cosseno de 0: ', cos(0))
```

Figura 20: Exemplo utilizando o *pyplot* com o nome *plt*



Fonte: Autores.

Neste exemplo, a primeira linha do código significa que foram importadas todas as funções disponíveis da biblioteca ***math***. A seguir, foi utilizada a função ***cos()*** para o cálculo do cosseno de zero. Também é possível realizar a importação de uma função com um nome específico escolhido pelo programador.

Por fim, mas não menos importante, destaca-se que o Python vem com "baterias inclusas". Ou seja, por padrão, existe uma vasta biblioteca de módulos e pacotes presentes no interpretador do Python.

De acordo com [1], alguns módulos importantes da biblioteca padrão, são:

- **Matemática:** *math*, *cmath*, *decimal* e *random*;
- **Sistema:** *os*, *glob*, *shutils* e *subprocess*;
- **Threads:** *threading*;
- **Persistência:** *pickle* e *cPickle*;
- **XML:** *xml.dom*, *xml.sax* e *elementTree* (a partir da versão 2.5);
- **Configuração:** *ConfigParser* e *optparse*;
- **Tempo:** *time* e *datetime*;
- **Outros:** *sys*, *logging*, *traceback*, *types* e *timeit*.

Para informações mais aprofundadas, fica como sugestão dos autores a lista das referências bibliográficas presente na última página desta apostila.

4 Desvios Condicionais (*if*, *elif* e *else*)

Nesta unidade, são apresentados os operadores condicionais. Isto é, os operadores que permitem o programa realizar verificações condicionais para uma lógica mais robusta e que proporcione melhores soluções ao problema a ser resolvido.

Em Python, os operadores condicionais são o ***if*** e o ***else***. A partir destes, existe a ramificação ***elif***, que é a junção do ***if*** com o ***elif*** para indentar os condicionais em sequência lógica.

```
a = 1
b = 2

if b > a:
    c = 'is cool! '
elif a == b:
    c = "isn't cool! "
else:
    c = "isn't cool! "
a = 'Apostila '
b = 'of Python '

print(a + b + c)
```

A saída desse código será:

```
# Apostila of Python is cool!
```

De forma intuitiva, a expressão que estiver após a palavra ***if*** é a expressão que será testada para a condição ser verdadeira ou falsa. Ou seja, caso a expressão resulte em ***True***, os comandos endentados após os dois pontos (***:***) serão executados; caso contrário, esses não serão executados. Se após o ***if*** existir o condicional ***else***, então tudo o que estiver indentado após o ***else*** será executado quando a expressão do ***if*** não o for.

Caso a lógica necessitar de condicionais sequenciais, que se não forem atendidos por uma expressão, devem ser testados em outra(s) expressão(ões), então pode-se utilizar os condicionais ***elif*** que são sinônimos abreviados de ***else if***. Vale pontuar que, ao se utilizar esses condicionais, deve-se atentar à lógica do programa, pois todos os ***elif*** estarão relacionados ao seus ***if*** anteriores, o que significa que todos os condicionais serão testados no caso do anterior resultar ***False***. Na prática a lógica fica mais intuitiva.

É importante destacar que, eventualmente é necessário utilizar algum condicional que não execute comando algum. Para isso, basta utilizar a instrução ***pass***, que não executa nenhum comando.

Além disso, pontua-se que os condicionais podem não conter o último comando ***else***. Nesse caso, se as condições dos ***if***'s e/ou ***elif***'s forem atendidas, o programa continua sua execução; caso contrário, nenhum comando será executado e o programa continua sua execução.

```
a = 1
b = 2
if b > a:
    if a < b:
        if b > a:
            print('\nb > a')
```

O código acima exemplifica o uso de vários condicionais indentados. Não há limites para o número de condicionais que podem ser endentados um dentro do outro.

Em Python, caso haja a necessidade de se verificar a condição de um intervalo ou de valores diferentes em relação a um terceiro, é possível simplificar sua sintaxe como segue no exemplo abaixo.

```
import random

a = random.random()
if -10 < a and a < 10:
    print('\n (-10, 10) é o intervalo de a.')
else:
    print('\n (-10, 10) não é o intervalo de a.')
```

Neste exemplo, a expressão condicional ***-10 < a and a < 10***, pode ser substituída por uma forma mais legível e elegante, como mostrado abaixo:

```
import random

a = random.random()
if -10 < a < 10:
    print('\n (-10, 10) é o intervalo de a.')
else:
    print('\n (-10, 10) não é o intervalo de a.')
```

5 Laços de Repetição

Nesta unidade, são abordados os laços de repetição (***for*** e ***while***), bem como algumas instruções comumente utilizadas nesses laços (***break*** e ***continue***).

5.1 Laço de repetição *for*

De acordo com [1], a instrução ***for*** é a estrutura de repetição mais utilizada em Python, sendo que essa aceita tanto sequências estáticas, quanto sequências geradas por iteradores.

Sua utilização é muito similar à de outras linguagens, como C, C#, entre outras, mudando-se apenas sua sintaxe. Abaixo é ilustrada uma forma simples de utilização do for.

```
for i in range(5):
    print(i)
```

Neste caso, o laço ***for*** imprime na tela o respectivo valor de ***i*** para cada iteração. O primeiro valor de ***i*** é ***0*** e segue sendo incrementado de um em um até o número ***4***.

Esta é uma das possíveis formas de se limitar o número de execuções do laço de repetição ***for***, porém existem inúmeras outras. O que deve ser primeiramente entendido é o raciocínio de sua utilização. Ou seja, para se utilizar o ***for*** é sempre necessário indicar uma variável iteradora que irá assumir um valor diferente para cada iteração do laço, e é sempre necessário indicar os limites de iteração. Em Python, as iterações do ***for*** podem ser realizadas de diversas formas. A variável iteradora pode assumir o valor de um número, *string* ou elemento de uma lista.

```
x = 'Apostila'
for i in x:
    print(i)
```

Como é possível observar (característica da linguagem), o laço ***for*** é muito intuitivo, ou seja, escrever ***for i in [0, 1, 2, 3] :*** é o mesmo que representar: *"Para cada valor de "i" dentro dos valores (0, 1, 2 e 3), faça:"*.

```
for i in [0, 1, 2, 3]:
    print(i)
```

Além disso, existem funções que podem ser utilizadas para limitar a execução do laço de repetição ***for***, como a função ***range(m, n, p)*** (utilizada anteriormente), que cria uma lista de inteiros começando em ***m*** e terminando em ***n - 1*** sendo incrementada de ***p*** em ***p***; e a função ***len()*** que retorna o tamanho de determinado vetor.

```
m = -1
n = +1
p = 1
for i in range(m, n, p):
    print(i)
```

Geralmente, a função ***len()*** é utilizada quando se quer percorrer uma lista, de tal forma que o valor da variável iteradora não assuma cada valor da lista e sim cada posição da lista. Ou seja, não importa o conteúdo de cada posição da lista, a variável iteradora irá assumir o valor de cada posição. O exemplo abaixo exemplifica isso.

Fica como desafio escrever ao lado deste exemplo qual o resultado final deste código.

```
x = ['Apostila', ' é', ' nota', 100]

for i in range(len(x)):
    print(i)

for i in x:
    print(i)
```

Além disso, o laço ***for*** pode ser melhor utilizado através das instruções ***break*** e ***continue***. Estas instruções também seguem o mesmo sentido de outras linguagens de programação. A instrução ***break*** interrompe o laço (terminando-o por completo) e a instrução ***continue*** pula para a próxima iteração imediatamente (não termina o laço, apenas passa à próxima iteração). O exemplo abaixo ilustra a utilização destas instruções.

```
for i in range(5):
    if i == 0:
        print('\ni = 0, Então: ', i)
    elif i == 1:
        print('\ni = 1, Então: continue')
        continue
    elif 1 < i < 3:
        print('\nA variável i, é: ', i)
    elif i == 3:
        print('\ni = 3, Então: break')
```

```
        break
    else:
        print('\ni > 3, Então: ', i)
```

A saída desse código será:

```
# i = 0, Então: 0
# i = 1, Então: continue
#  A variável i, é: 2
# i = 3, Então: break
```

Destaca-se que no último exemplo, a instrução ***continue*** poderia ser substituída pela instrução condicional ***pass***, sem prejuízo algum.

5.2 Laço de Repetição *while*

Outro laço de repetição comumente utilizado em Python é o ***while***. Este também é muito semelhante ao de outras linguagens. De forma simples, o laço é executado enquanto sua condição for verdadeira. Sua sintaxe em Python é demonstrada na abaixo, como segue.

```
a = 0
b = 2
while a <= b:
    print('\n', a, ' <= ', b, ' ')
    a += 1
```

Em que a saída será:

```
# 0 <= 2
# 1 <= 2
# 2 <= 2
```

Uma utilização muito comum do laço ***while*** é para se criar laços infinitos para a modelagem e criação de jogos. Como o ***while*** executa todas suas instruções caso a expressão condicional for verdadeira, então para se criar um laço infinito basta que essa expressão seja sempre verdade. Para isso, pode-se escrever ***while 1==1:*** ou, de forma mais direta e lógica, ***while True:***. O exemplo abaixo ilustra essa utilização.

```
i = 0
while True:
    print(i)
```

```
    i += 1
```

Em que a saída será:

```
# 3046131
# 3046132
# 3046133
# 3046134
# 3046135
# 3046136
# 3046137
# 3046138
# 3046139
# .......
```

No exemplo acima, é possível observar que o programa continua somando ***1 + 1*** infinitamente e, a cada soma, o resultado é impresso na tela. Neste caso, a instrução ***break*** poderia ser utilizada para encerrar as repetições através de condicionais (if, elif, entre outros), bem como a instrução ***continue***, para passar à próxima iteração.

Segundo [1], o laço de repetição ***while*** é adequado quando não se sabe quantas iterações devem ocorrer até se atingir um objetivo específico e quando não há uma sequência a ser seguida.

6 *Strings* e Conjuntos de Dados

Nesta unidade, são apresentados vetores, as matrizes e *strings* em Python.

6.1 *Strings*

Na unidade 3.5 foram mencionadas algumas informações sobre strings em Python. Nesta seção, estas *builtins* (funções nativas presentes no interpretador sem precisarem ser importadas) são abordadas de forma mais ampliada.

Segundo [1], *strings* são imutáveis, ou seja, não é possível adicionar, remover ou substituir seus caracteres sem que outra *string* seja criada. O autor de [3] conceitua uma *string* como uma sequência/coleção ordenada de caracteres.

Existem dois tipos de *strings*, representadas no exemplo abaixo.

- **String Unicode** → s = s'Python';
- **String Padrão** → s = 'Python'

O *Unicode* fornece um número único (*code point*) para cada caractere, independentemente da plataforma, programa ou linguagem. Funciona como um padrão mundial de conversão de caracteres. Seus primeiros 127 *code points* são compatíveis com os códigos da tabela ASCII. Uma *string* padrão pode ser convertida para *Unicode* fazendo uso da função ***unicode()***.

Como visto anteriormente, a inicialização de uma *string* pode ser tanto com aspas simples, como com aspas duplas (ou normais). Como *strings* são sequências fixas de caracteres, cada caractere está alocado na memória ocupando uma posição. Dessa forma, é possível acessar cada caractere de uma *string* acessando sua respectiva posição através da utilização de colchetes *[]*. O código abaixo ilustra um exemplo de acesso aos caracteres específicos de uma *string*.

```
>>> s = "Python top!"
>>> s[0]
'P'
>>> s[0 : 5]
'Python'
>>> s[-4 : -1]
'top'
```

No exemplo acima foi utilizado o fatiamento de uma *string*. Isso pode ser muito útil em diversas aplicações e pode ser entendido de forma intuitiva lembrando que cada caractere (incluindo espaços) ocupam uma posição no espaço destinado à *string*. Então, para fatiar ou selecionar apenas uma parcela desses espaços, basta utilizar o operador ***[m : n]***, em que ***m*** e ***n*** são inteiros e representam os limites de fatiamento da *string*. Neste processo é considerado que a fatia da *string* vai de ***m*** até ***n - 1***. Caso queira acessar os elementos finais da *string*, pode-se utilizar os índices negativos. Nesse raciocínio, o índice ***-1*** representa o último caractere, o ***-2*** representa o penúltimo e assim por diante.

Como uma *string* é uma sequência, então ela pode ser percorrida por laços de repetição (***for*** ou ***while***). Abaixo é apresentado um exemplo para percorrer uma *string* utilizando os laços ***for*** e ***while***. Fica como desafio o(a) leitor(a) escrever ao lado do exercício qual será o resultado deste código (as duas formas resultam na mesma saída).

```
## Utilizando o laço while:
s = "Python top!"
i = 0
while i != len(s):
    i += 1
    print(s[-i])

## Utilizando o laço for:
s = "Python top!"
for j in range(len(s)):
    print(s[-j-1])
```

Além disso, *strings* podem ser concatenadas utilizando o operador ***+*** ou podem ser interpoladas utilizando o operador ***%*** junto da letra que corresponde ao tipo da variável considerada. A Tabela 2 mostra as letras utilizadas nos tipos diferentes de variáveis.

Por fim, destaca-se que *strings* possuem métodos próprios que são muito úteis em inúmeros exemplos. Alguns dos mais utilizados são ***upper()*** que eleva todos os caracteres da *string* para maiúsculos, ***find()*** que busca um determinado caractere dentro da *string* e retorna sua posição, ***count()*** que conta o número de repetições de um caractere dentro da string e ***split()*** que recorta a *string*, transformando-a em uma lista [4]. Abaixo é ilustrada a utilização de alguns destes métodos.

```
>>> s = 'Apostila de python top!'
>>> print('\nA string "' + s + '" em caixa alta, é: ', s.upper())
```

[4]Outros métodos podem ser facilmente encontrados em: https://docs.python.org/3/library/string.html

```
A string "apostila de python top!" em caixa alta, é: APOSTILA DE PYTHON
                                        TOP!
>>>
>>> print('\nA posição do caractere "!", é: ', s.find('!'))

A posição do caractere "!", é: 22
>>>
>>> print('\nO número de repetições do caractere "i", é: ', s.count('i'
                                       ))

O número de repetições do caractere "i", é: 1
>>>
>>> print('\nA string recortada em todas as letras "o", é: ', s.slpit('
                                       o'))

A string recortada em todas as letras "o", é: ['ap', 'stila de pyth', '
                                       n t', 'p!']
```

6.2 Listas, Tuplas e Dicionários

Esta é uma das seções mais importantes da programação básica em Python. Como esta é uma apostila introdutória, os conteúdos de Listas, Tuplas e Dicionários não serão esgotados (pois para cada um seria necessário um capítulo inteiro), porém nesta seção são pontuados os pilares básicos de cada uma dessas sequências.

6.2.1 Listas

Em Python, uma lista é uma sequência mutável de n valores que podem ser de qualquer tipo (inclusive outras listas, tupas e dicionários). De forma simples, uma lista pode ser entendida como um vetor de elementos que podem ser de qualquer tipo. As listas são exatamente iguais às *strings*, exceto pelo fato de *strings* serem imutáveis e listas serem mutáveis.

Abaixo é apresentado um exemplo de sintaxe de uma lista:

```
>>> lista = ['Pode conter qualquer tipo de valor',0 , 5 + 8j, ['Outras
                                       listas', 'por exemplo'], 5.5]
>>> lista
['Pode conter qualquer tipo de valor',0 , (5+8j), ['Outras listas', '
                                       por exemplo'], 5.5]
```

Também é possível realizar a criação de uma lista vazia com o objetivo de preenche-la posteriormente. Para isso, basta atribuir à variável escolhida o abre e fecha colchetes ***lista_vazia = []***.

As listas podem ser percorridas, fatiadas e concatenadas da mesma forma que as *strings*. A diferença é que em se tratando de *strings*, cada elemento é um caractere e, em se tratando de listas, cada elemento pode ser qualquer tipo de variável. Além disso, uma *string* pode ser convertida para uma lista (como já foi visto) e uma lista pode ser convertida para uma *string*. Abaixo são ilustradas estas duas conversões.

```
>>> a = 'Apostila de Python é top!'
>>> # Converter cada letra: list()
. . .
>>> b = list(a)
>>> b
['A', 'p', 'o', 's', 't', 'i', 'l', 'a', ' ', 'd', 'e', ' ', 'P', 'y',
                                    't', 'h', 'o', 'n', 'é', ' ', 't',
                                    'o', 'p', '!']
>>>
>>> #Converter por espaços ou por elementos previamente indicados:
                                     split()
. . .
>>> b = a.split()
>>> b
['Apostila', 'de', 'Python', 'é', 'top!']
>>>
>>> # Converter lista para string: join()
. . .
>>> # join() concatena todos os elementos da lista. Caso a lista não
                                    possua espaços posicionados de
                                    forma correta, então é necessário
                                    que o join() seja utilizado a
                                    partir de um ' ' (espaço) ou de uma
                                     variável com a mesma finalidade.
. . .
>>> a = ' '.join(b)
>>> a
'Apostila de Pyhton é top!'
>>> # Ou:
. . .
>>> espaço = ' '
>>> a = espaço.join(b)
>>> a
'Apostila de Python é top!'
```

Listas em Python possuem alguns métodos e algumas operações que auxiliam e muito o trabalho do programador. A seguir serão mostrados alguns exemplos de

utilização destes métodos e dessas operações [5].

```
>>> lista = ['Pode conter qualquer tipo de valor',0, 5 + 8j, ['Outras
                                    listas', 'por exemplo'], 5.5]
>>> # Para acrescentar um novo elemento após o último elemento:
. . .
>>> lista.append('Novo elemento')
>>> lista
['Pode conter qualquer tipo de valor',0 , (5+8j), ['Outras listas', '
                                    por exemplo'], 5.5, 'Novo elemento'
                                    ]

>>> # Para excluir um elemento "n" da lista:
. . .
>>> lista.remove(5.5)
>>> lista
['Pode conter qualquer tipo de valor', 0, (5+8j), ['Outras listas', '
                                    por exemplo'], 'Novo elemento']
>>>
>>> # Para ordenar todos os elementos da lista em ordem ascendente:
                                    sort() obs.: todos os elementos
                                    devem ser do mesmo tipo
. . .
>>> lista = [-1, 50, 9, 7, 0, -6, 100]
>>> lista.sort()
>>> lista
[-6, -1, 0, 7, 9, 50, 100]
>>>
>>> # Para inverter todos os elementos da lista: reverse()
. . .
>>> lista = ['Pode conter qualquer tipo de valor', 0, 5 + 8j, ['Outras
                                    listas', 'por exemplo'], 'Novo
                                    elemento']
>>> lista.reverse()
>>> lista
[5.5, ['Outras listas', 'por exemplo'], (5+8j), 0, 'Pode conter
                                    qualquer tipo de valor']
>>>
>>> # Para retirar o último elemento da lista: pop() obs.: este método
                                    retorna o elemento retirado.
. . .
>>> print('O elemento retirado da lista, é: ', lista.pop())
O elemento retirado da lista, é: Pode conter qualquer tipo de valor
>>> lista
[5.5, ['Outras listas', 'por exemplo'], (5+8j), 0]
```

[5] Para obter mais informações sobre outros métodos e operações, sugerimos a consulta ao material disponibilizado em [1, 3] e a documentação do Python no link: https://docs.python.org/2/tutorial/datastructures.html.

É interessante destacar que os métodos ***sort()*** e ***pop()*** são muito utilizados aos se trabalhar com estruturas de dados do tipo Fila e Pilha.

6.2.2 Tuplas

Tuplas são muito semelhantes às listas, exceto pelo fato de que tuplas são imutáveis. Ou seja, as operações mostradas na seção anterior, onde foi possível mudar cada elemento da lista, como por exemplo ***lista[0] = qualquer coisa*** não são possíveis em tuplas. De forma concisa, tuplas não permitem apagar, acrescentar e nem realizar atribuições a tuplas já criadas. Abaixo é mostrado a sintaxe de uso de uma tupla.

```
>>> tupla = (0, 1, 2, 'muito parecio com listas')
>>> tupla
(0, 1, 2, 'muito parecido com listas')
>>> tupla = 0, 1, 2, 'muito parecio com listas'
>>> tupla
(0, 1, 2, 'muito parecido com listas')
>>> # Parenteses são opcionais.
```

Para criar-se uma Tupla com apenas um elemento, deve-se escrever: ***nome_da_variável = (elemento)***. É necessário se escrever a vírgula pois caso contrário o interpretador irá entender que a variável ***nome_da_variável*** é apenas uma variável do mesmo tipo do ***elemento***.

Além disso, é possível se converter listas em tuplas. Um exemplo desta operação é apresentada no exemplo abaixo.

```
>>> # Lista para Tupla:
. . .
>>> lista = [0, 1, 2, 3]
>>> tupla = tuple(lista)
>>> tupla
(0, 1, 2, 3)
>>>
>>> # Tupla para Lista:
. . .
>>> lista = list(tupla)
>>> lista
[0, 1, 2, 3]
```

O mesmo pode acontecer para uma *string*. Basta, de forma intuitiva, utilizar as funções ***tuple()*** ou ***list()***. A única ressalva que deve ser mencionada é que, para

converter de tupla para *string*, o método ***join()*** não pode ser utilizado, então pode-se utilizar a função ***str()***, porém essa função irá transformar tudo o que representa a tupla (inclusive os parênteses) em uma *string unicode*. Para melhor se converter de um tipo para outro, deve-se converter de tupla para lista e então para *string* (com o uso da função ***join()***).

Um exemplo de utilização prática de uma tupla pode ser para se separar endereços de *login* dos seus respectivos domínios (*hotmail*, *outlook*, entre outros). Abaixo é ilustrada essa operação.

```
>>> email = 'apostila@muitoTop.com'
>>> u_name, dominio = email.split("@")
>>> print('\nNome do usuário: ', u_name, '\nNome do domínio: ', dominio
                                        )

Nome do usuário: minicurso
Nome do domínio: muitoTop.com
```

Embora uma tupla seja imutável, esta pode ser formada por elementos mutáveis (como listas) e então esses elementos podem ser atualizados, desde que se respeite o formado da Tupla. O exemplo presente abaixo ilustra esse caso.

```
>>> tupla = [0, 1], 'a', 'b'
>>> # Neste caso, o primeiro elemento da tupla é uma lista (mutável).
                                    Então para se modificar esse
                                    elemento, deve-se acessar a posição
                                     desse elemento na tupla e então
                                    utilizar os métodos da lista. Não
                                    se pode atribuir uma nova lista ao
                                    elemento da Tupla.
. . .
>>> tupla[0].appen(2)
>>> tupla
([0, 1, 2]. 'a', 'b')
>>>
>>> # Jeito errado:
. . .
>>> tupla[0] = [0, 1, 2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

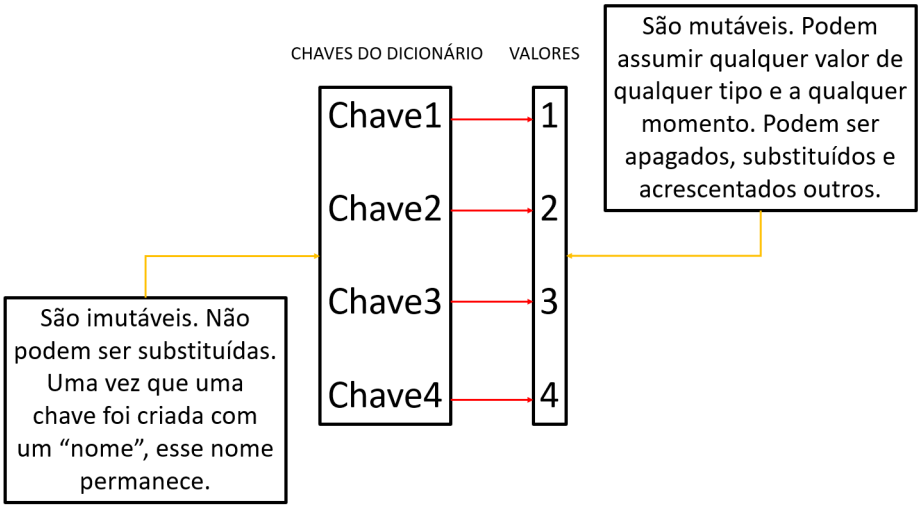
Segundo [1], de forma geral as tuplas são mais eficientes que as listas pois aquelas consomem menos recursos computacionais (memória), por serem estruturas mais simples.

6.2.3 Dicionários

Segundo [1], um dicionário é uma lista de associações composta por uma chave (de tipo imutável) e estruturas correspondentes às chaves que podem ser mutáveis ou não. Isso caracteriza os dicionários como mutáveis (pois os valores em si são mutáveis, exceto as chaves). Segundo [3], os dicionários são um dos melhores recursos do Python. Abaixo é apresentado um exemplo de uso de dicionários.

```
>>> # Forma geral:
. . .
>>> dicionario = {'chave1' : 1, 'chave2' : 2, 'chave3' : 3}
>>> dicionario
{'chave1' : 1, 'chave2' : 2, 'chave3' : 3}
>>>
>>> # Outra forma:
. . .
>>> dicionario = dict()
>>> dicionario['chave1'] = 1
>>> dicionario['chave2'] = 2
>>> dicionario['chave3'] = 3
>>> dicionario
{'chave1' : 1, 'chave2' : 2, 'chave3' : 3}
```

Figura 21: Formação dos dicionários



Fonte: Autores.

Além disso, um dicionário pode ser criado vazio seguindo o mesmo raciocínio de listas e das tuplas. Para isso, basta escrever: ***nome_da_variavel = {}***. A

seguir, para adicionar novos elementos (com suas respectivas chaves) ao dicionário, basta escrever: ***nome_da_variavel['nome_da_chave'] = elemento a ser adicionado***. Embora as chaves dos dicionários sejam imutáveis, essas chaves não precisam ser, necessariamente, uma *string*. Em muitos casos, as chaves podem ser tuplas (que também são imutáveis).

A seguir são pontuados alguns métodos e funções geralmente utilizadas com dicionários. Para maiores informações e para se obter outros métodos e funções, recomenda-se consultar as bibliografias citadas anteriormente e a documentação do Python que pode ser acessada através do link: https://docs.python.org/2/tutorial/datastructures.html#dictionaries.

```
>>> tel = {'John' : 456789, 'Peter' : 784785, 'Vini' : 987584, 'James'
                                    : 142365, 'Mary' : 456987}
>>> # Adicionar um novo número à agenda:
. . .
>>> tel['Paul'] = 101010
>>> tel
{'John': 456789, 'Peter': 784785, 'Vini' : 987584, 'James': 142365, '
                                     Mary': 456987, 'Paul': 101010}
>>>
>>> # Verificar se existe uma chave no dicionário:
. . .
>>> 'Paul' in tel
True
>>>
>>> # Retirar um telefone da agenda:
. . .
>>> del tel['James']
>>> tel
{'John': 456789, 'Peter': 784785, 'Vini' : 987584, 'Mary': 456987, '
                                     Paul': 101010}
>>>
>>> # Método que mostra os itens com suas chaves de um dicionário:
. . .
>>> tel.items()
dict_items([('John', 456789), ('Peter', 784785), ('Vini', 987584), ('
                                     Mary', 456987), ('Paul', 101010)])
>>>
>>> # Método que mostra todas as chaves de um dicionário:
. . .
>>> tel.keys()
dict_keys(['John', 'Peter', 'Vini', 'Mary', 'Paul'])
>>>
>>> # Método que mostra todos os valores presentes em um dicionário:
. . .
>>> tel.values()
dict_values([456789, 784785, 456987, 101010])
```

```
>>>
>>> # A função builtin "len()" retorna o tamanho da lista (neste caso,
                                        o número de chaves):
. . .
>>> print(len(tel))
4
```

Por fim, vale destacar que em Python ainda existem outras funções que geram estruturas padrão consideradas como sequências. São estas: ***set()*** e ***frozenset()***. A função ***set()*** cria uma sequência mutável, não ordenada e unívoca (exclui todos os elementos repetidos). Já a função ***frozenset()*** cria uma sequência imutável, não ordenada e de igual forma unívoca.

Ambas criam sequências, porém proporcionam a implementação de operações de conjuntos, como por exemplo: intersecção, diferença e união.

6.3 Matrizes

Em Python, não existe uma função padrão que trabalhe com matrizes. Entretanto, é relativamente fácil e intuitivo a criação destas. Nesse caso, a melhor opção para se trabalhar com matrizes de uma forma segura é através das inúmeras bibliotecas e módulos disponíveis para esse fim. Dessa forma, destaca-se a biblioteca ***NumPy***, que é uma biblioteca projetada com códigos de alta performance. Sua instalação é simples e esse conteúdo foi abordado na seção sobre Bibliotecas.

Contudo, se o objetivo é utilizar uma matriz de forma mais simples, é possível implementar uma matriz através da criação de uma lista, cujos elementos são outras listas. Isto é, para a criação de uma matriz quadrada ***n*** (n X n), basta que seja criada uma lista com ***n*** elementos e dentro de cada um seja criada outra lista com ***n*** elementos.

Existem diversas formas de se fazer isso. Abaixo é ilustrada uma dessas formas juntamente com o resultado impresso na tela.

```
mat = []
linhas = 5
colunas = 5
for i in range(linhas):
    lin = []
    for j in range(colunas):
        lin.append(int(i))
    mat.append(lin)
```

```
for i in mat: print(i)


## RESULTADO:
[0, 0, 0, 0, 0]
[1, 1, 1, 1, 1]
[2, 2, 2, 2, 2]
[3, 3, 3, 3, 3]
[4, 4, 4, 4, 4]
```

Além dessa forma, abaixo é mostrado outro jeito de se criar uma matriz. Desta forma, utiliza-se menos linhas e é mais eficiente pois é utilizada a compreensão de listas (***List Comprehension***) [6]. O resultado é exatamente igual ao do exemplo anterior.

```
# Utilizando comprensão de listas (List Comprehension)
mat1 = [[i for j in range(5)] for i in range(5)]
for i in mat1: print(i)
```

Outro fato sobre matrizes em Python é que, por ser uma lista de outras listas, todos os métodos e funções das listas podem ser utilizados em matrizes (de forma correta). Cada elemento pode ser obtido a partir da posição bidimensional ***matriz[linha][coluna]***.

[6] Mais informações podem ser obtidas em: https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions

7 Funções

Nesta unidade, será apresentado o conteúdo relacionado à criação de funções, como essas são definidas e utilizadas em Python. De acordo com [3], uma função é uma sequência nomeada de instruções que executa uma determinada operação, sendo criada a partir de um nome e de instruções que a compõe, possibilitando ser utilizada posteriormente.

De acordo com [1], as funções, em Python:

- Podem retornar ou não objetos;
- Aceitam *Doc Strings*;
- Se não for passado o parâmetro será igual ao *default* definido na função. Portanto aceitam parâmetros *default*;
- Aceitam que os parâmetros sejam passados com nome. Neste caso, a ordem em que os parâmetros foram passados não importa;
- Tem *namespace* próprio (escopo local), e por isso podem ofuscar definições de escopo global;
- Podem ter suas propriedades alteradas (geralmente por decoradores).

Abaixo é ilustrada a sintaxe da declaração de uma função.

```
def nomeDaFuncao(parametro):
    """Esta é a forma de se declarar
    funções em Python. Este texto é um
    Doc String."""
    print('Esta função recebeu ', parametro, ' como parâmetro. '
                    '\nO tipo deste parâmetro, é: ',
                    type(parametro))
    return '\nApostila de Python é top!'

print(nomeDaFuncao('string'))
```

A utilização desta função é mostrada abaixo:

```
## RESULTADO:
Esta função recebeu string como parâmetro.
O tipo deste parâmetro, é: <class 'str'>

Apostila de Python é top!
```

A partir do momento em que uma função é criada, é possível utilizá-la dentro de outras funções e, inclusive, dentro dela mesma. Esse conceito é base para recursividade de funções. A seguir é mostrado um exemplo de uma função recursiva que calcula o fatorial de um número ***n***.

```
def fatorial(numero):
    if type(numero) is not int:
        return '\nO número deve ser um inteiro!'
    if numero <= 0:
        return 1
    else:
        return numero * fatorial(numero-1)
```

Percebe-se que a função fatorial foi reutilizada no resultado da primeira chamada da função, o que a caracteriza como uma função recursiva. É interessante pontuar que, em muitos casos, é do interesse do programador criar funções que não retornem nada, apenas façam algo ou imprimam algo na tela. Nesse caso, essas funções são chamadas de funções nulas, segundo [3]. Para criar uma função que não retorna nada, basta escrever no final da função ***return None***, ou simplesmente não escrever a instrução ***return***.

```
def naoRetornaNada_ApenasPrinta():
    print('\nApenas printa qualquer coisa')

naoRetornaNada_ApenasPrinta()
```

Além disso, os argumentos recebidos nas funções também podem ser listas, tuplas e/ou dicionários (além dos valores *default*). Para listas, basta escrever ***** na frente do nome do argumento na função; neste caso, todos os parâmetros presentes após o asterisco serão considerados como uma lista de elementos. Porém, o tipo de ***return*** será uma tupla. Abaixo é mostrada essa afirmação.

```
def recebe_lista(*lista):
    return lista

a = recebe_lista(0, 5, 8, 7, 6, 4)
print(a)
print('TIPO: ', type(a))

## RESULTADO:
(0, 5, 8, 7, 6, 4)
TIPO: <class 'tuple'>
```

Para criar um dicionário com base em uma lista de argumentos, basta utilizar ****** na frente do nome do argumento na função. Neste caso, cada valor no argumento

deve ser indicado utilizando o operador = para indicar chaves e valores. Abaixo é apresentado um exemplo de uso.

```
def recebe_dicionario(**dic):
    return dic

a = recebe_dicionario(a=1, b=2, c=3)
print(a)
print('TIPO: ', type(a))

## RESULTADO:
{'a': 1, 'b': 2, 'c': 3}
TIPO: <class 'dict'>
```

Para valores padrão, a função aceita (mas não exige) que um valor seja adicionado à função no momento de sua instanciação. Caso esse valor não seja indicado, a função assumirá o valor padrão indicado a ela. Abaixo é ilustrada essa afirmação.

```
def funcaoDefault(a, b=5):
    c = a + b
    return '\n' + str(a) + ' + ' + str(b) + ' = ' + str(c)

i = funcaoDefault(7)  # Não foi indicado o valor de "b". Então esse
                                    assume seu valor default b = 5.
print('\n', i)
j = funcaoDefault(7, 7)  # Agora "b" recebeu o valor de 7.
print('\n', j)
l = funcaoDefault(b=10, a=100)  # Também é possível declarar qual o
                                    valor de cada argumento da função.
print('\n', l)
```

O resultado do código acima é mostrado abaixo.

```
## RESULTADO:
7 + 5 = 12


7 + 7 = 14


100 + 10 = 110
```

Por fim, faz-se necessário destacar que o fluxo da programação em Python sempre é interrompido no momento que uma função é chamada. Quando isso acontece, ele interrompe a sequência do código, executa a função invocada e, após o término da função, retorna a sequência do ponto em que havia parado.

Por esse motivo, é necessário que as funções sejam previamente criadas. Assim, recomenda-se que essas estejam localizadas após a importação das bibliotecas e/ou módulos utilizados, no início do código.

Para maiores informações sobre os conteúdos abordados nesta apostila ou para outros, recomenda-se os autores de [1, 3], além da documentação do Python que pode ser acessada em: https://docs.python.org/3/.

8 Programação Orientada a Objetos

Neste capítulo, são mostrados alguns dos principais fundamentos que sustentam a *OOP - Object Oriented Programming* ou *Programação Orientada a Objetos*, que é um paradigma de programação muito usual e presente em praticamente todas as linguagens de programação, principalmente em Python, a qual é fortemente orientada a objetos, conforme visto no Capítulo 1.

A programação orientada a objetos tem suas raízes na década de 60 e, em consenso, Alan Kay foi um dos pioneiros neste segmento. A seguir, encontra-se uma de suas frases.

> *"I invented the term Object-Oriented, and I can tell you I did not have C++ in mind."* - Alan Kay

Assim, este paradigma de programação fornece algumas vantagens, das quais pode-se citar como principais:

- Reúso de códigos;
- Sintetização de códigos;
- Incremento da produtividade;
- Melhor flexibilidade do sistema;
- Facilidade na manutenção;
- Divisão de responsabilidades.

Entretanto, segundo [4], existem 4 pilares básicos que caracterizam uma programação orientada a objetos. A seguir são mostrados, de forma sucinta, essas 4 bases.

8.1 Abstração

A Abstração é um pilar extremamente importante à orientação a objetos, pois é através dos conceitos englobados neste tópico que cada objeto é caracterizado.

Isso se deve ao fato de que ao lidar com objetos reais, faz-se necessário a abstração desses objetos, ou seja, é necessário entender o que é esse objeto e o que esse faz/proporciona à programação.

Este tópico pode ser subdividido em três partes, as quais caracterizam a abstração. São elas:

8.1.1 Identidade

É o que nomeia o objeto.

```
class Ponto:
    """
    Esta é a forma de se criar uma Classe
    em Python, cujo nome é "Ponto".
    Cada classe possui um nome único e é
    convencionado que os nomes de classes
    possuam a primeira letra maiúscula.

    Neste caso, esta classe dá origem a um objeto
    "Ponto", que é um ponto bidimensional.
    """
```

8.1.2 Propriedades

É o que caracteriza o objeto, em relação ao que esse possui.

```
class Ponto:
    def __init__(self, coordenadaX, coordenadaY):
        """
        Este é o método denominado "construtor".
        É aqui onde estão contidos todos os atributos
        deste objeto. Neste caso, o objeto "Ponto"
        deve receber os parâmetros "coordenadaX" e
        "coordenadaY" toda vez que for instanciado.

        :param coordenadaX: Pode ser entendido como
        a posição X do ponto bidimensional.

        :param coordenadaY: Pode ser entendido como
        a posição Y do ponto bidimensional.
        """
        self.x = coordenadaX  # self.x recebe o valor do parâmetro
                                    coordenadaX
        self.y = coordenadaY  # self.y recebe o valor do parâmetro
                                    coordenadaY
```

Destaca-se que a palavra ***self*** é obrigatória ao se referir ao objeto em si dentro da classe.

8.1.3 Métodos

É o que indica o que este objeto faz.

```
def setX(self, novoX):
    """
    Este método "seta" um novo valor para a posição x.

    :param novoX: Este é o novo valor que se deseja atualizar
    ao ponto.
    :return: Neste caso, por se tratar de um método "getter and setters
                                                    ",
    não há retorno.
    """
    self.x = novoX  # A variável self.x é atualizada.

    def setY(self, novoY):
        """
        Este método "seta" um novo valor para a posição y.

        :param novoY: Este é o novo valor que se deseja atualizar
        ao ponto.
        :return: Neste caso, por se tratar de um método "getter and
                                                    setters",
        não há retorno.
        """
        self.y = novoY  # A variável self.y é atualizada.
```

Assim, tem-se uma classe criada. a qual possui uma identidade (nome), atributos (o que tem nela) e métodos (o que ela faz).

```
class Ponto:
"""
Esta é a forma de se criar uma Classe
em Python, cujo nome é "Ponto".
Cada classe possui um nome único e é
convencionado que os nomes de classes
possuam a primeira letra maiúscula.

Neste caso, esta classe dá origem a um objeto
"Ponto", que é um ponto bidimensional.
"""

    def __init__(self, coordenadaX, coordenadaY):
        """
        Este é o método denominado "construtor".
        É aqui onde estão contidos todos os atributos
        deste objeto. Neste caso, o objeto "Ponto"
        deve receber os parâmetros "coordenadaX" e
```

```
        "coordenadaY" toda vez que for instanciado.

        :param coordenadaX: Pode ser entendido como
        a posição X do ponto bidimensional.

        :param coordenadaY: Pode ser entendido como
        a posição Y do ponto bidimensional.
        """
        self.x = coordenadaX  # self.x recebe o valor do parâmetro
                                            coordenadaX
        self.y = coordenadaY  # self.y recebe o valor do parâmetro
                                            coordenadaY

    def setX(self, novoX):
        """
        Este método "seta" um novo valor para a posição x.

        :param novoX: Este é o novo valor que se deseja atualizar
        ao ponto.
        :return: Neste caso, por se tratar de um método "getter and
                                            setters",
        não há retorno.
        """
        self.x = novoX  # A variável self.x é atualizada.

    def setY(self, novoY):
        """
        Este método "seta" um novo valor para a posição y.

        :param novoY: Este é o novo valor que se deseja atualizar
        ao ponto.
        :return: Neste caso, por se tratar de um método "getter and
                                            setters",
        não há retorno.
        """
        self.y = novoY  # A variável self.y é atualizada.
```

8.2 Encapsulamento

O encapsulamento é outra das principais bases que sustentam a programação orientada a objetos. Esta é uma das partes que agregam segurança à programação pois proporcionam a possibilidade de esconder determinadas propriedades, ou de forma mais formal, possibilitam a distinção de elementos públicos, protegidos ou

privados.

Sobre isso, pontua-se em [4]:

> *"A maior parte das linguagens orientadas a objetos implementam o encapsulamento baseado em propriedades privadas, ligadas a métodos especiais chamados getters e setters, que irão retornar e setar o valor da propriedade, respectivamente. Essa atitude evita o acesso direto a propriedade do objeto, adicionando uma outra camada de segurança à aplicação."*

De forma geral, encapsulamento pode ser entendido como a parte responsável por proporcionar a distinção entre as partes internas e externas de um objeto.

Em Python, as três formas de se encapsular os elementos são mostradas abaixo.

- Public:

```
class Ponto:
    def __init__(self, coordenadaX, coordenadaY):
        self.x = coordenadaX  # self.x é uma variável pública
        self.y = coordenadaY  # self.y é uma variável pública

    def setX(self, novoX):
        self.x = novoX  # A variável self.x é atualizada.

    def setY(self, novoY):
        self.y = novoY  # A variável self.y é atualizada.

# Neste caso, não é necessário utilizar os métodos ``setX"
# e ``setY" porque mesmo fora da classe Ponto é possível
# acessar as variáveis ``self.x" e ``self.y".

ponto1 = Ponto(0, 0)
ponto1.x = 1  # Não foi necessário utilizar o método ``setX"
ponto1.y = 1
```

- Protected:

```
class Ponto:
    def __init__(self, coordenadaX, coordenadaY):
        self._x = coordenadaX  # self.x é uma variável protegida
        self.y = coordenadaY  # self.y é uma variável pública

    def setX(self, novoX):
        self._x = novoX  # A variável self.x é atualizada.

    def setY(self, novoY):
        self.y = novoY  # A variável self.y é atualizada.
```

```
ponto1 = Ponto(0, 0)
ponto1._x = 1  # Como esta atualização está sendo feita
# fora da classe, esta linha de comando não faz sentido.
ponto1.y = 1  # Ao contrário, a variável ``y" continua
# pública, sendo possível atualizá-la fora da classe
# sem o uso de um método específico.
```

- Private:

```
class Ponto:
    def __init__(self, coordenadaX, coordenadaY):
        self.__x = coordenadaX  # self.x é uma variável privada
        self.y = coordenadaY  # self.y é uma variável pública

    def setX(self, novoX):
        self.__x = novoX  # A variável self.x é atualizada.

    def setY(self, novoY):
        self.y = novoY  # A variável self.y é atualizada.

ponto1 = Ponto(0, 0)
ponto1.__x = 1  # Como esta atualização está sendo feita
# fora da classe, esta linha de comando não faz sentido.
ponto1.y = 1  # Ao contrário, a variável ``y" continua
# pública, sendo possível atualizá-la fora da classe
# sem o uso de um método específico.

# Neste caso, para se atualizar a variável ``X", é necessário utilizar
                                    o método ``setX".
ponto1.setX(1)
```

8.3 Herança

A herança é outro conceito bastante difundido na programação orientada a objetos. Isso se dá pelo fato de que através deste conceito é possível reutilizar blocos de códigos e, literalmente, herdar características de outros objetos.

Segundo [3]:

> A herança é a capacidade de definir uma nova classe que seja uma versão modificada de uma classe existente.

A seguir, encontra-se um exemplo de uma classe que herda as características de outra. Neste caso, a classe **Linha** herda as características da classe **Ponto** para se criar uma sequência de pontos que dão origem a uma linha.

```
class Ponto:
def __init__(self, coordenadaX, coordenadaY):
    self.x = coordenadaX  # self.x recebe o valor do parâmetro
                                        coordenadaX
    self.y = coordenadaY  # self.y recebe o valor do parâmetro
                                        coordenadaY

def setX(self, novoX):
    self.x = novoX  # A variável self.x é atualizada.

def setY(self, novoY):
    self.y = novoY  # A variável self.y é atualizada.

def mostra(self):
    return '(', self.x, ', ', self.y, ') '


class Linha(Ponto):  # Esta é a sintaxe para se herdar uma classe.
    # Neste caso, a classe Linha está herdando a classe Ponto
    def __init__(self, coordenadaX, coordenadaY):
        super().__init__(coordenadaX, coordenadaY)
        self.__linha = []

    def addPonto(self, x, y):
        ponto = Ponto(x, y)
        self.__linha.append(ponto)


linha1 = Linha(0, 0)
linha1.addPonto(1, 1)
linha1.addPonto(2, 2)
linha1.addPonto(3, 3)
```

8.4 Polimorfismo

Neste texto, o último pilar que sustenta o paradigma de orientação a objetos é o polimorfismo. Esse conceito é oriundo de analogias feitas com a natureza, pois nessa são encontrados animais capazes de alterar suas características em determinadas situações, conforme a necessidade.

Em se tratando de objetos, o autor de [4] explica esta definição, como segue:

> Como sabemos, os objetos filhos herdam as características e ações de seus "ancestrais". Entretanto, em alguns casos, é necessário que as ações para um mesmo método seja diferente. Em outras palavras, o polimorfismo consiste na alteração do funcionamento interno de um método herdado de um objeto pai.

Como exemplo, pode-se citar um objeto "pai"(ancestral comum) denominado *Veiculo*. Esse objeto possui os métodos *setPassageiros*, *setVelocidade* e *setCombustivel*. A partir desse objeto, outros dois objetos são instanciados *Carro* e *Aviao*, os quais herdam as características do objeto *Veiculo*.

Entretanto, em ambos os casos a herança das características são realizadas de forma diferente. Isto é, o objeto *Carro* irá utilizar os método *setPassageiros*, *setVelocidade* e *setCombustivel* de forma diferente da forma como o objeto *Aviao* irá utilziar. O que caracteriza o polimorfismo.

Assim, encontram-se os quatro pilares essenciais na compreensão de uma linguagem orientada a objetos. Cada linguagem utiliza esses conceitos de forma singular, porém na maioria dos casos o raciocínio é o mesmo.

Referências

[1] Luiz Eduardo Borges. *Python para desenvolvedores: aborda Python 3.3*. Novatec Editora, 2014.

[2] CodinGame. Why learn python. Online; accessed 08 jun. 2019.

[3] Allen Downey. *Think Python*. "O'Reilly Media, Inc.", 2012.

[4] DevMedia Henrique. Os 4 pilares da programação orientada a objetos, 2014. Online; accessed 24 jul. 2019.

[5] Medium. Advantages and disadvantages of python programming language, 2017. Online; accessed 08 jun. 2019.

[6] PythonBrasil. Ides para python. Online; accessed 08 jun. 2019.

[7] DataFlair Team. Python applications – 9 real world applications of python programming, 2018. Online; accessed 08 jun. 2019.

[8] Trytoprogram. History of python programming. Online; accessed 08 jun. 2019.