



Introdução à Linguagem Python

Paradigmas de Linguagens de Programação

Mariana Cossetti Dalfior
Ausberto S. Castro Vera

21 de junho de 2023



Disciplina: Paradigmas de Linguagens de Programação 2023

Linguagem: Python

Aluno: Mariana Cossetti Dalfior

Ficha de avaliação:

Aspectos de avaliação (requisitos mínimos)	Pontos
Introdução (Máximo: 01 pontos) <ul style="list-style-type: none"> • Aspectos históricos • Áreas de Aplicação da linguagem 	
Elementos básicos da linguagem (Máximo: 01 pontos) <ul style="list-style-type: none"> • Sintaxe (variáveis, constantes, comandos, operações, etc.) • Cada elemento com exemplos (código e execução) 	
Aspectos Avançados da linguagem (Máximo: 2,0 pontos) <ul style="list-style-type: none"> • Sintaxe (variáveis, constantes, comandos, operações, etc.) • Cada elemento com exemplos (código e execução) • Exemplos com fonte diferenciada (listing) 	
Mínimo 5 Aplicações completas - Aplicações (Máximo : 2,0 pontos) <ul style="list-style-type: none"> • Uso de rotinas-funções-procedimentos, E/S formatadas • Uma Calculadora • Gráficos • Algoritmo QuickSort • Outra aplicação • Outras aplicações ... 	
Ferramentas (compiladores, interpretadores, etc.) (Máximo : 1,0 pontos) <ul style="list-style-type: none"> • Ferramentas utilizadas nos exemplos: pelo menos DUAS • Descrição de Ferramentas existentes: máximo 5 • Mostrar as telas dos exemplos junto ao compilador-interpretador • Mostrar as telas dos resultados com o uso das ferramentas • Descrição das ferramentas (autor, versão, homepage, tipo, etc.) 	
Organização do trabalho (Máximo: 01 ponto) <ul style="list-style-type: none"> • Conteúdo, Historia, Seções, gráficos, exemplos, conclusões, bibliografia • Cada elemento com exemplos (código e execução, ferramenta, nome do aluno) 	
Uso de Bibliografia (Máximo: 01 ponto) <ul style="list-style-type: none"> • Livros: pelo menos 3 • Artigos científicos: pelo menos 3 (IEEE Xplore, ACM Library) • Todas as Referências dentro do texto, tipo [ABC 04] • Evite Referências da Internet 	
Conceito do Professor (Opcional: 01 ponto)	
<p style="text-align: right;">Nota Final do trabalho:</p>	

Observação: Requisitos mínimos significa a metade dos pontos

Copyright © 2023 Mariana Cossetti Dalfior e Ausberto S. Castro Vera

UENF - UNIVERSIDADE ESTADUAL DO NORTE FLUMINENSE DARCY RIBEIRO

CCT - CENTRO DE CIÊNCIA E TECNOLOGIA

LCMAT - LABORATÓRIO DE MATEMÁTICAS

CC - CURSO DE CIÊNCIA DA COMPUTAÇÃO

Primeira edição, Abril, 2023

Sumário

1	Introdução	7
1.1	Aspectos históricos da linguagem Python	7
1.2	Áreas de Aplicação da Linguagem	10
1.2.1	Big Data	10
1.2.2	Orientação a objetos	10
1.2.3	Desenvolvimento Web	11
1.2.4	Análise de Dados	12
2	Conceitos básicos da Linguagem Python	13
2.1	Variáveis	13
2.1.1	Regras de nomenclatura de uma variável	13
2.2	Constantes	14
2.3	Entrada e Saída de Dados	15
2.3.1	Input	15
2.3.2	Output	16
2.4	Tipos de Dados Básicos	17
2.4.1	String	17
2.4.2	Operações lógicas	21
2.4.3	Números Inteiros	23
2.4.4	Números de Ponto Flutuante	25
2.4.5	Números complexos	27
2.4.6	Booleanos	28
2.5	Estrutura de Controle e Funções	30
2.5.1	O comando <i>If</i>	30
2.5.2	O comando <i>Else</i>	31
2.5.3	O comando <i>Elif</i>	32

2.5.4	Laço <i>For</i>	33
2.5.5	Laço <i>While</i>	34
3	Aspectos Avançados da Linguagem Python	37
3.1	Classes	37
3.1.1	Classes base Abstratas	37
3.2	Tipos de Dados de Coleção	37
3.2.1	Tipos Sequenciais	38
3.2.2	Tipos Conjunto	41
3.2.3	Tipos Mapeamento	43
3.2.4	Metaprogramação de classes	46
3.2.5	Heranças	48
3.3	Funções Geradoras	51
4	Aplicações da Linguagem Python	53
4.1	Operações básicas	53
4.2	Programas com Objetos	56
4.3	O algoritmo Quicksort	58
4.4	Aplicações com Banco de Dados	59
4.5	Programa de Cálculo Numérico	60
5	Ferramentas existentes e utilizadas	65
5.1	PyCharm	65
5.2	Visual Studio Code	66
6	Conclusão	69
	Bibliografia	72



1. Introdução

De acordo com [Tul16], a linguagem de programação Python foi idealizada inicialmente por Guido van Rossum no início dos anos 1990. Ela é sucessora da linguagem ABC e possui seu nome inspirado em um grupo de comédia chamado Monty Python. Python é uma linguagem interpretada, o que significa que ela utiliza um intérprete que permite ler o código linha por linha sem a necessidade de compilar o código completo de uma vez. É considerada uma linguagem de alto nível, assim como PHP, Java, Fortran e C, porém, possui como diferencial a sua sintaxe clara e concisa, a rapidez de processamento e uma maior regularidade que as outras linguagens de alto nível, tornando-se mais fácil de ler e aprender.

Além de ser uma linguagem de programação orientada a objetos, o que significa que os programas são construídos em torno de objetos que podem ser determinados e manipulados pelo programador, Python também é uma linguagem de multiplataforma. Isso significa que ela pode ser executada em diversos sistemas operacionais, como Windows, MacOS e Linux, possibilitando que os desenvolvedores elaborem programas capazes de serem executados em diferentes plataformas sem que ocorram grandes modificações no código.

Python apresenta uma ampla diversidade de bibliotecas, dentre elas destacam-se Pandas, Numpy, Matplotlib, Pillow, Scikit-learn e TensorFlow. Devido à facilidade e rapidez proporcionada pela grande variedade de bibliotecas e módulos, essa linguagem é amplamente empregada em Data Science, IA (Inteligência Artificial), Machine Learning (Aprendizado de Máquina), automação, desenvolvimento web e outros campos.

1.1 Aspectos históricos da linguagem Python

A história da linguagem de programação Python vem sendo desenvolvida desde a sua criação no início da década de 1990 até os dias atuais, a qual foi se tornando cada vez mais popular.

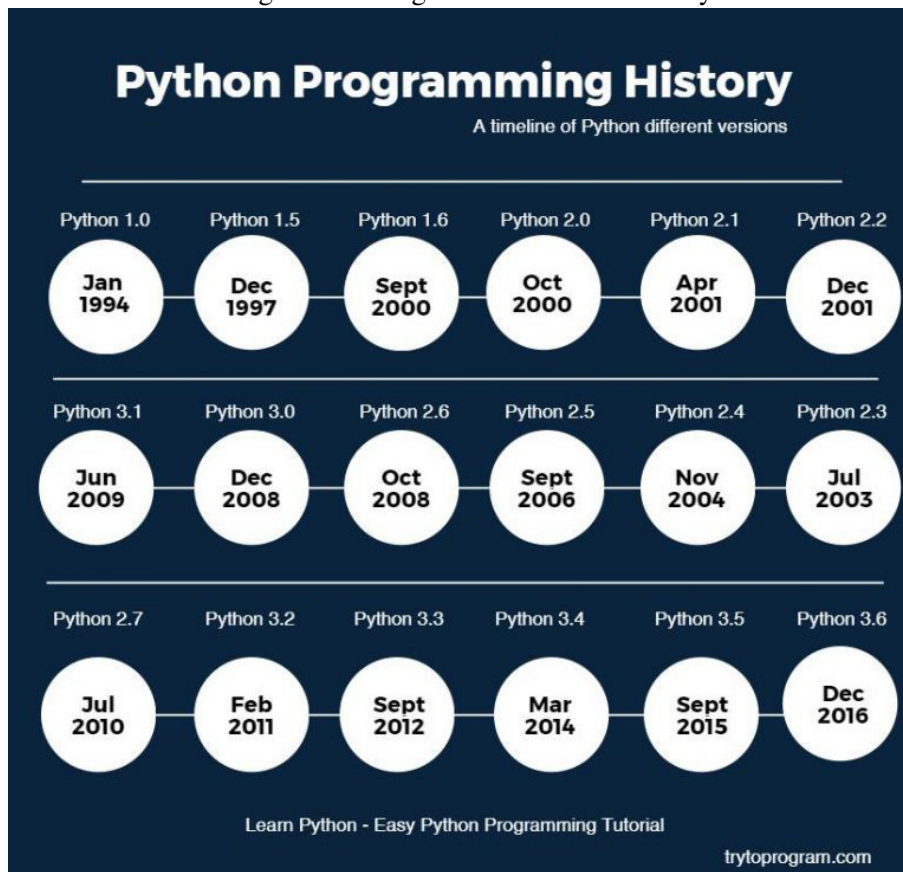
Relata-se a seguir alguns aspectos históricos dessa linguagem. As informações exibidas logo a seguir foram baseadas em (<https://docs.python.org/3/whatsnew/index.html>), [vR10, dSS19, Mon12] :

- O principal autor da linguagem Python foi o Guido van Rossum.

- Divergente do que acreditam, Python não obteve esse nome devido a espécie de serpente Píton. Ele foi nomeado assim em homenagem a um seriado de comédia. *Monty Python's Flying Circus* de que Guido van Rossum era fã.
- Python conquistou uma certa popularidade em áreas específicas, como processamento de texto e computação científica, durante a década de 1990.
- No ano 1991, foi lançada a primeira versão pública do Python o 0.9.0. Nela foi incluída suporte para a manipulação de estruturas de controle de fluxo, strings e funções. Além da inclusão de algumas características de programação funcional.
- No ano 1994, van Rossum lançou a versão 1.0 do Python, que adicionou novos recursos e melhorias à linguagem, como suporte para orientação a objetos e para módulos e pacotes.
- Guido lançou a versão 1.2 do Python antes de desligar-se do seu emprego na CWI
- Em 1995, Guido lançou novas versões do Python enquanto trabalhava em uma nova empresa CNRI (Corporation for National Research Initiatives) em Reston, nos Estados Unidos. Uma delas foi a versão 1.4, a qual foi implementada o suporte nativo a números complexos e parâmetros nomeados.
- Python 1.6 foi a última versão a ser lançada na CNRI, no ano de 2000.
- Em 2000, também foi lançada a versão Python 2.0 na BeOpen, entretanto foi o único lançamento nessa empresa. Implementando um sistema de coletor de lixo com capacidade de tratar ciclos de referências e também a list comprehension (Compreensão de listas).
- Python ganhou bastante popularidade do início de 2000, com o uso crescente de Python em Data Science e Machine Learning.
- A versão Python 2.1 era semelhante as versões 1.6.1 e 2.0, a qual recebeu as mudanças na especificação que suporte escopo aninhado e foi lançada em 2001.
- Também em 2001 a versão Python 2.2 lançada e possuiu uma grande novidade, em que houve a união dos tipos Python - que eram escritos em C - e as classes - que já eram escritas em Python - em uma só hierarquia.
- Em 2008 foi publicado Python 2.6, já com a intenção de fazer a transição dele para o Python 3.0. Uma nova maneira de formatar strings e bibliotecas para multiprocessamento foi incluída na versão 2.6.
- Também em 2008 foi lançada Python 3.0, que introduziu mudanças na sintaxe e manipulação de strings. Tornando Python mais fácil de aprender, porém isso gerou algumas incompatibilidades com versões mais antigas.
- No ano de 2010, foi criado um framework Django para facilitar o desenvolvimento de aplicações web na linguagem de programação Python.
- O PEP 484, foi aprovado em 2015, possibilitando adicionar tipos opcionais ao Python. Essa mudança foi projetada com o intuito de ajudar os desenvolvedores a escreverem um código mais confiável e seguro. Nesse ano também foi lançada a versão Python 3.5.
- Entre 2015 e 2023 foram lançadas as versões desde a Python 3.6 até Python 3.11. Nessas

novas versões foram adicionadas melhorias no interpretador, na sintaxe, velocidade, biblioteca padrão e etc.

Figura 1.1: Progressão das versões do Python



Fonte: www.tryprogram.com

A seguir é possível observar algumas logotipos da linguagem Python que foram desenvolvidas com o passar dos anos:

Figura 1.2: Logos da Linguagem Python



Fonte: www.pngwing.com

Dessa forma, conclui-se que o Python é uma linguagem de programação versátil e popular que é utilizada em diversas áreas, que vão desde IA (Inteligência Artificial) até a análise de dados. Como possui uma sintaxe simplista e legível a linguagem torna-se fácil de aprender e usar, fornecendo aos programadores uma série de ferramentas e recursos proporcionada pela grande variedade de

bibliotecas e estruturas. Como Python está se tornando cada vez mais popular provavelmente prosseguirá a interpretar um papel relevante na área da tecnologia e do software.

1.2 Áreas de Aplicação da Linguagem

Python é uma linguagem de programação muito versátil que pode ser aplicada em diversas áreas da tecnologia como, por exemplo big data, orientação a objetos, desenvolvimento web e análise de dados.

1.2.1 Big Data

De acordo com [McK18], a linguagem de programação Python é altamente utilizada no campo da big data por possuir a capacidade de tratar uma larga escala de dados e de realizar o processamento paralelo. Além de dispor inúmeras bibliotecas e frameworks para o tratamento de big data ela é uma linguagem de fácil aprendizado.

As principais áreas de big data em que são utilizados o Python:

- **Análise de dados:** Python é utilizado para a análise de dados por dispor inúmeras bibliotecas de análise de dados. As principais utilizadas nessa área são Pandas, NumPy, SciPy, Matplotlib, que possibilitam os desenvolvedores a realizar procedimentos como processamento de dados em larga escala, análise estatística e visualização de dados.
- **Machine Learning (Aprendizado de Máquina):** Tratando-se de uma das mais utilizadas linguagens de programação para a elaboração de modelos de aprendizado de máquina. Python possui TensorFlow, Scikit-learn e PyTorch como bibliotecas famosas de machine learning.
- **Processamento de dados em tempo real:** Proporcionando aos programadores um processamento de larga escala de dados e uma resposta mais veloz a eventos em tempo real, as ferramentas como Apache Storm e Apache Kafka são para o processamento de dados em tempo real e amplamente utilizadas em Python.
- **Processamento de dados distribuídos:** As ferramentas de sistemas de processamento de dados como Apache Spark e Apache Hadoop de Python possibilitam que programadores processem em larga escala conjuntos de dados em clusters distribuídos de computadores.

Desse modo, devido a possuir uma vasta diversidade de bibliotecas e frameworks, flexibilidade e também ser de fácil uso, a linguagem de programação Python é uma das principais escolhas para ser utilizada em big data, por existir uma demanda por big data em contínuo crescimento, Python provavelmente prosseguirá se tornando uma escolha cada vez mais popular em big data.

1.2.2 Orientação a objetos

Entre um mar de linguagens de programação orientadas a objetos (POO), o Python, que possui a orientação a objetos como parte essencial da linguagem, se destaca fornecendo suporte completo para esse tipo de programação. De acordo com [Fel19, Fel20], a programação orientada a objetos é dedicada à criação de objetos com atributos e métodos, organizando-os em classes, hierarquia e heranças, servindo assim como um meio de desenvolvimento de software. Além do mais, o Python permite que desenvolvedores criem códigos mais modulares e versáteis, através do polimorfismo em que as classes podem executar o mesmo método de diferentes maneiras. Ainda, a linguagem de programação Python possui uma biblioteca padrão poderosa em POO chamada "pickle", que é

utilizada para fazer a desserialização e serialização de objetos e outra chamada "abc" para definir interfaces e classes abstratas.

Alguns conceitos da orientação a objetos em Python:

- Classes - A classe é usada para a criação de objetos e definir seus métodos e atributos.

Exemplo: A classe chamada 'Pessoa' recebe os atributos: 'nome' e 'idade'.

- Herança - A herança em Python é utilizada para a criação de novas classes com características como atributos e métodos, herdadas de classes que já existem.

Exemplo: Uma nova classe 'Faculdade' herda os atributos da classe 'Aluno', podendo ser adicionados novos atributos nessa classe como, 'curso' e 'matrícula'.

- Hierarquia - Na linguagem de programação Python, é possível criar uma hierarquia de classes, ou seja, uma classe é a "principal" em relação a outras classes.

Exemplo: A classe 'Cachorro' é a principal em relação a classe como 'raça', a qual recebe atributos 'nome' da classe principal, além de ter seus próprios atributos 'cor' e 'porte'.

- Polimorfismo - Com o polimorfismo é possível criar um método em uma classe, podendo ser executado de diversas formas.

Exemplo: Temos a classe 'Pessoa' e outras classes que herdam dela: 'Aluno' e 'Professor', as duas possuem um método chamado 'matricular' porém cada uma implementa de uma forma diferente.

Dessa forma, Python é uma linguagem boa que suporta programação orientada a objetos, utilizada frequentemente em desenvolvimento de aplicativos web, desenvolvimentos de jogos, desenvolvimento de aplicativos desktop e etc. Escolhido principalmente por possuir um sintaxe simplista e de fácil aprendizagem.

1.2.3 Desenvolvimento Web

Python é uma linguagem de programação amplamente utilizada no desenvolvimento web, especialmente devido a ter uma vasta diversidade de bibliotecas e frameworks. Os frameworks mais utilizados para o desenvolvimento web são Flask e Django, porém existem outros frameworks em Python que são importantes para o desenvolvimento web, como o Pyramid.

De acordo com [?], os Frameworks em Python utilizados em desenvolvimento web:

- Django - Um dos frameworks web mais utilizados em Python. Usando o Django criar sites ou aplicativos web escaláveis e seguros se torna muito fácil, pois com ele é possível a criação completa desses em pouco tempo, por possuir muitos recursos integrados, como o gerenciamento de URL, autenticação de usuários e a administração de banco de dados.
- Flask - Um framework web de aprendizado mais fácil que o Django. O Flask é usado para desenvolver aplicações web com incomplexidade e rapidez, possuindo como principal característica a facilidade em criar rotas e funcionalidades web.
- Pyramid - O framework web Pyramid para Python é o que possui maior flexibilidade e extensividade em relação ao Django e ao Flask. Visto que, ele é apropriado para o desenvolvimento de grandes e complexos aplicativos web, por oferecer autenticação, cache, segurança e internacionalização que são uma grande diversidade de recursos.

Portanto, a linguagem de programação Python é utilizado em diversas áreas do desenvolvimento web, que vão desde a construção de aplicativos web até a análise de dados. Python oferece

uma grande diversidade de bibliotecas e frameworks que atendem a todas as necessidades de desenvolvimento web, sem depender do tamanho e da dificuldade do projeto.

1.2.4 Análise de Dados

A linguagem Python possui uma diversidade de bibliotecas disponíveis e facilidade de uso tornando-a bem popular em análise de dados - que consiste em coletar, processar e analisar dados para adquirir informações relevantes e insights sobre um determinado tema, sendo utilizada em diferentes áreas, como saúde, marketing e finanças. As bibliotecas Pandas, NumPy, Matplotlib e SciPy são as mais populares de Python que são empregues nessa área.

De acordo com [McK23, Che18], a análise de dados em Python abrange um conjunto de etapas, a primeira etapa na análise de dados em Python é coletar os dados relevantes para o problema em questão. Após realizar a coleta desses dados, é fundamental prepará-los para uma análise. Nesta, é possível ser incluído o preenchimento de dados ausentes, a limpeza de dados, regularização de dados e a transformação desses em formatos apropriados para a análise. Logo após é realizado uma análise exploratória de dados (EDA), que é uma técnica a qual possui como finalidade compreender o grupo de dados, examinando a sua distribuição, identificando os prováveis padrões, as tendências e os vínculos entre as variáveis. Além da EDA é realizada a análise estatística que é um procedimento para conhecer as relações entre variáveis do conjunto de dados. Podendo ser incluídas a análise de correlações, regressão e outras técnicas estatísticas. Após essas análises, ocorre a modelagem preditiva, que abrange a criação de modelos estatísticos para prever efeitos futuros com base nos dados à disposição. A última etapa da análise de dados em Python realiza a apresentação das descobertas da análise de dados de maneira clara e sucinta.

Python em análise de dados realiza um processo iterativo envolvendo diversas etapas, que vão desde a coleta de dados até o relatório dos resultados. Alguns exemplos de áreas em que é largamente usada:

- Finanças: Utilizada para realizar a análise de risco, previsão de preços e gestão de investimentos.
- Saúde: Usada para prever predisposições de saúde e reconhecer tratamentos mais eficazes.

Finda-se então que a análise de dados em Python está em um constante evolução, tornando-se poderosa na extração de informações custosas de um amplo conjunto de dados. Possuindo ferramentas capazes de realizar tarefas desde um simples processamento de dados até análises profundas de Machine Learning (Aprendizado de Máquina).



2. Conceitos básicos da Linguagem Python

Neste capítulo serão apresentados os aspectos básicos da linguagem de programação Python, incluindo atribuições, variáveis, funções, estruturas de controle, input/output e tipos de dados.

2.1 Variáveis

As variáveis são elementos essenciais na programação em Python, visto que são espaços reservados na memória que armazenam dados - desde número e textos, até objetos - que podem ser alterados durante a execução de um programa. Em Python segundo [eS20] as variáveis são tratadas como objetos, em que cada objeto receberá um determinado valor a ser armazenado ou um resultado obtido através de um comando. Para a criação de uma variável é necessário escolher um nome e atribuir a ela um valor utilizando '='. Exemplos de variáveis em Python:

```
>>> # variavel com str
>>> Nome = 'Mariana Cossetti Dalfior'

>>> # variavel com int
>>> Idade = 20

>>> # variavel com float
>>> Peso = 55
```

2.1.1 Regras de nomenclatura de uma variável

Existem algumas regras para nomear as variáveis em Python.

O que é permitido:

- Começar com uma letra ou com um sublinhado '_ '.
- O restante do nome pode receber letras, números e sublinhados.
- Os nomes das variáveis são sensíveis a maiúsculas e minúsculas. Então 'teste' e 'Teste' são nomes distintos de variáveis.

```
>>> # Permitido
>>> nome
>>> Idade
>>> idade
>>> _bolo
>>> nota_1
```

O que não é permitidos:

- Começar com número.
- Palavras-chave da linguagem utilizadas para funções, como `'if'`, `'else'`, `'while'`, `'for'`, `'def'`, `'class'` não podem ser empregues como nomes de variáveis.

```
>>> # Nao permitido
>>> 1bolo
>>> 25_anos
>>> for
>>> %juros
```

2.2 Constantes

Na linguagem de programação Python não existe um tipo específico para as constantes como acontece em outras linguagens. Entretanto, de acordo com [eS20] é possível realizar a criação de constantes através de uma convenção de nomenclatura, utilizando o nome da constante escrito em letras maiúsculas e no lugar dos espaços utilizar o `'_'`. Porém, essa convenção não impossibilita que o valor da variável seja modificado durante a execução do programa, mas faz com que os programadores entendam e tratem essa variável como constante. Exemplos de constantes em Python:

```
>>> # Valor de Pi
>>> PI = 3.141592

>>> # Constante de Napier (base do logaritmo natural)
>>> NAPIER = 2.71828
```

Ademais, existem módulos que possuem constantes já pré-definidas em Python, como o `'math'` que dispõem de *Pi* e a constante de *Euler*. Como acessar as constantes desse módulo são mostrados a seguir com o código fonte 2.1 e o resultado 2.2 dos exemplos em Python:

```
>>> # importando biblioteca
>>> import math
>>>
>>> # mostra pi
>>> print('O valor de Pi: {}'.format(math.pi))
O valor de Pi: 3.141592653589793
>>>
>>> # mostra euler
>>> print('O valor de Euler: {}'.format(math.e))
O valor de Euler: 2.718281828459045
```

Figura 2.1: Código fonte do exemplo de constantes

```
Constantes.py
1  # importando biblioteca
2  import math
3
4  # mostra pi
5  print('O valor de Pi: {}'.format(math.pi))
6
7  # mostra euler
8  print('O valor de Euler: {}'.format(math.e))
```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 2.2: Resultado do código fonte do exemplo de constantes

```
PS C:\Mariana - UENF\PLP\Python\Códigos>
tantes.py"
O valor de Pi: 3.141592653589793
O valor de Euler: 2.718281828459045
```

Fonte: Criado por Mariana Cossetti Dalfior

2.3 Entrada e Saída de Dados

Segundo [Yoo22] a entrada e saída de dados em Python é realizada através da utilização de uma biblioteca padrão.

2.3.1 Input

A entrada de dados é feita por meio da função `'input()'`, permitindo que seja inserido valores ou informações - sejam elas strings, números e booleanos - pelos usuários enquanto o programa é executado. Essa função solicita que o usuário digite uma informação - a execução do programa fica pausada até que essa informação seja retornada - e então a armazena em uma variável. Porém todos os dados que são inseridos usando essa função são tratados como strings, então para transformá-los em outro tipo será necessário a utilização do `'int()'` ou `'float()'`. A seguir é possível observar o código fonte 2.3 e o resultado 2.4 dos exemplos em Python:

```
>>> # entrada em formato de str
>>> aluno = input('Nome: ')
>>>
>>> # entrada em formato de int
>>> matricula = int(input('Numero de matricula: '))
>>>
>>> # entrada em formato de float
>>> nota = float(input('Nota da prova: '))
>>>
```


Figura 2.3: Código fonte do exemplo de input

```
Input.py > ...
1  # entrada em formato de str
2  aluno = input('Nome: ')
3
4  # entrada em formato de int
5  matricula = int(input('Numero de matricula: '))
6
7  # entrada em formato de float
8  nota = float(input('Nota da prova: '))
```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 2.4: Resultado do código fonte do exemplo de input

```
PS C:\Mariana - UENF\PLP\Python\Códigos>
t.py"
Nome: Mariana
Numero de matricula: 20211100064
Nota da prova: 10
```

Fonte: Criado por Mariana Cossetti Dalfior

2.3.2 Output

A saída de dados em Python é comumente executada pela função `'print()'`, possuindo como encargo exibir na tela as informações geradas pelo programa para o usuário - seja ela uma mensagem, valores e informações. EA seguir é possível observar o código fonte ?? e o resultado 2.6 dos exemplos de saída de dados em Python:

```
>>> # saida de uma frase
>>> print('Um prato de trigo para tres tigres tristes.')
Um prato de trigo para tres tigres tristes.

>>> # saida do valor de uma variavel
>>> aluno = "Mariana"
>>> matricula = 20211100064
>>> print('{} tem a matricula {}'.format(aluno, matricula))
Mariana tem a matricula 20211100064
```


Figura 2.5: Código fonte do exemplo de output

```
Output.py > ...
1  # saída de uma frase
2  print('Um prato de trigo para tres tigres tristes.')
3
4  # saída do valor de uma variavel
5  aluno = "Mariana"
6  matricula = 20211100064
7  print('{} tem a matricula {}'.format(aluno, matricula))
```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 2.6: Resultado do código fonte do exemplo de output

```
PS C:\Mariana - UENF\PLP\Python\Códigos> &
ut.py"
Um prato de trigo para tres tigres tristes.
Mariana tem a matricula 20211100064
```

Fonte: Criado por Mariana Cossetti Dalfior

No último exemplo foi utilizado o `'format()'`, utilizado para substituir os espaços que foram reservados para as variáveis fornecidas como argumento.

2.4 Tipos de Dados Básicos

Os tipos de dados básicos são utilizados para realizar as operações básicas de acordo com [MRA19]. Esses podem ser divididos nos seguintes tipos de dados:

2.4.1 String

As strings em Python são uma parte fundamental da linguagem, pois representam uma sequência de caracteres - podem ser letras, símbolos, letras e espaços em branco. Elas são criadas utilizando aspas simples e duplas. A seguir é possível observar o código fonte 2.7 e o resultado 2.8 dos exemplos de strings em Python:

```
>>> # Usando aspas simples
>>> exemStr = 'Vasco da Gama'
>>> print (exemStr)
Vasco da Gama

>>> # Usando aspas duplas
>>> exemStr2 = "Vasco da Gama o maior do Rio!"
>>> print (exemStr2)
Vasco da Gama o maior do Rio!
```

Figura 2.7: Código fonte do exemplo de strings

```
String1.py > ...
1  # Usando aspas simples
2  exemStr = 'Vasco da Gama'
3  print (exemStr)
4
5  # Usando aspas duplas
6  exemStr2 = "Vasco da Gama o maior do Rio!"
7  print (exemStr2)
```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 2.8: Resultado do código fonte do exemplo de strings

```
PS C:\Mariana - UENF\PLP\Python\Códigos>
ng1.py"
Vasco da Gama
Vasco da Gama o maior do Rio!
```

Fonte: Criado por Mariana Cossetti Dalfior

Em Python as strings suportam diversos métodos embutidos - que servem para a formatação e manipulação das mesmas - como `.upper()` e `.lower()`. A seguir é possível observar o código fonte 2.9 e o resultado 2.10 dos exemplos de aplicação desses métodos:

```
>>> # Usando o .upper()
>>> exemStr = 'Gabriel Pec'
>>> print(exemStr.upper())
GABRIEL PEC

>>> # Usando o .lower()
>>> exemStr2 = 'Leo Pele'
>>> print(exemStr2.lower())
leo pele
```

Figura 2.9: Código fonte do exemplo de `.upper` e `.lower`

```
String2.py > ...
1  # Usando o .upper()
2  exemStr = 'Gabriel Pec'
3  print(exemStr.upper())
4
5  # Usando o .lower()
6  exemStr2 = 'Leo Pele'
7  print(exemStr2.lower())
```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 2.10: Resultado do código fonte do exemplo de `.upper` e `.lower`

```
PS C:\Mariana - UENF\PLP\Python\Códigos>
ng2.py"
GABRIEL PEC
leo pele
```

Fonte: Criado por Mariana Cossetti Dalfior

A seguir é possível observar o código fonte 2.11, 2.13 e o resultado 2.12, 2.14 dos exemplos de manipulação de Strings utilizadas em Python:

- *Concatenação de strings*

Duas ou mais strings podem ser concatenadas utilizando o operador `'+'`.

```
>>> # Concatenando 2 strings
>>> print ("Cruz" + " de Malta")
Cruz de Malta
```

Figura 2.11: Código fonte do exemplo de concatenação de strings

```
Concatenacao1.py
1 # Concatenando 2 strings
2 print ("Cruz" + " de Malta")
```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 2.12: Código fonte do exemplo de concatenação de strings

```
PS C:\Mariana - UENF\PLP\Python\Códigos>
atenacao1.py"
Cruz de Malta
```

Fonte: Criado por Mariana Cossetti Dalfior

Outra forma de juntar strings e de forma mais eficiente é através do uso do método `'join()'`, capaz de concatenar um lista de strings em apenas uma string.

```
>>> # Concatenando 2 nomes
>>> jogadores = ["Jair", "Andrey"]
>>> print( ", ".join(jogadores))
Jair, Andrey
```

Figura 2.13: Código fonte do exemplo de concatenação de lista de strings

```
concatenacao2.py > ...
1 # Concatenando 2 nomes
2 jogadores = ["Jair", "Andrey"]
3 print( ", ".join(jogadores))
```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 2.14: Resultado do código fonte do exemplo de concatenação de lista de strings

```
PS C:\Mariana - UENF\PLP\Python\Códigos>
atenacao2.py"
Jair, Andrey
```

Fonte: Criado por Mariana Cossetti Dalfior

- *Operador de indexação*

Qualquer elemento individual de uma lista, string ou tupla pode ser acessado com o operador de indexação '[]', usando apenas o índice entre os colchetes depois da variável que armazena essa sequência. Existem duas formas de indexar os caracteres de um string em Python:

Index com inteiros positivos começando da esquerda para direita utilizando o 0 como index do primeiro caractere da sequência.

Index com inteiros negativos começando da direita para a esquerda utilizando o -1 para iniciar o index sendo ele o último elemento da sequência, -2 o penúltimo elemento, e assim consecutivamente.

A seguir é possível observar o código fonte 2.15 e o resultado 2.16 do exemplo de indexação em Python:

```
>>> # indexando com inteiros positivos
>>> frase = "Vasco da Gama"
>>> print(frase[0:5])
Vasco
```

Figura 2.15: Código fonte do exemplo de indexação

```
frase.py > ...
1 # indexando com inteiros positivos
2 frase = "Vasco da Gama"
3 print(frase[0:5])
```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 2.16: Resultado do código fonte do exemplo de indexação

```
PS C:\Mariana - UENF\PLP\Python\Códigos>
e.py"
Vasco
```

Fonte: Criado por Mariana Cossetti Dalfior

- *Formatação de strings*

Strings podem ser formatadas de forma dinâmica com base em valores variáveis através do método '.format()'. Esse método permite a criação de strings com espaço reservado para serem preenchidos com valores variáveis. A seguir é possível observar o código fonte 2.17, 2.19 e o resultado 2.18, 2.20 do exemplo da formatação em Python:

```
>>> nome = "Pedro Raul"
>>> idade = 26
```

```
>>> print("O jogador {} possui {} anos." .format(nome, idade))
O jogador Pedro Raul possui 26 anos.
```

Figura 2.17: Código fonte do exemplo de formatação de strings

```
Indexacao.py > ...
1 nome = "Pedro Raul"
2 idade = 26
3 print("O jogador {} possui {} anos." .format(nome, idade))
```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 2.18: Resultado do código fonte do exemplo de formatação de strings

```
PS C:\Mariana - UENF\PLP\Python\Códigos>
xacao.py"
O jogador Pedro Raul possui 26 anos.
```

Fonte: Criado por Mariana Cossetti Dalfior

Em versões mais atuais do Python existe outra tipo de formatação, utilizando uma forma mais intuitiva e simples.

```
>>> nome = "Pedro Raul"
>>> idade = 26
>>> print("O jogador", nome, "possui", idade, "anos.")
O jogador Pedro Raul possui 26 anos.
```

Figura 2.19: Código fonte do outro exemplo de formatação de strings

```
Format2.py > ...
1 nome = "Pedro Raul"
2 idade = 26
3 print("O jogador", nome, "possui", idade, "anos.")
```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 2.20: Resultado do código fonte do outro exemplo de formatação de strings

```
PS C:\Mariana - UENF\PLP\Python\Códigos>
at2.py"
O jogador Pedro Raul possui 26 anos.
```

Fonte: Criado por Mariana Cossetti Dalfior

2.4.2 Operações lógicas

Os operadores lógicos são utilizados para realizar operações sobre dados booleanos(lógicos). Os resultados dessas operações podem retornar *true*(verdadeiro) ou *false*(falso). A seguir serão

mostrados quais são esses operadores e o que eles significam.

Operações	Operador
Menor que	<
Maior que	>
Menor ou igual	<=
Maior ou igual	>=
Igual	==
Diferente de	!=

Tabela 2.1: Operações lógicas em Python

A seguir é possível observar o código fonte [2.21](#) e o resultado [2.22](#) do exemplo da utilização de operações lógicas:

```
>>> # Menor que
>>> print(35 < 12)
False
>>> # Maior que
>>> print(64 > 40)
True
>>> # Menor ou igual
>>> print(120 <= 120)
True
>>> # Maior ou igual que
>>> print(32 >= 165)
False
>>> # Igual
>>> print(15 == 80)
False
>>> # Diferente de
>>> print(30 != 15)
True
```

Figura 2.21: Código fonte do exemplo de operações lógicas

```
OperacoesLogicas.py
1  # Menor que
2  print(35 < 12)
3
4  # Maior que
5  print(64 > 40)
6
7  # Menor ou igual
8  print(120 <= 120)
9
10 # Maior ou igual que
11 print(32 >= 165)
12
13 # Igual
14 print(15 == 80)
15
16 # Diferente de
17 print(30 != 15)
```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 2.22: Resultado do código fonte do exemplo de operações lógicas

```
PS C:\Mariana - UENF\PLP\Python\Códigos> .\OperacoesLogicas.py
False
True
True
False
False
True
```

Fonte: Criado por Mariana Cossetti Dalfior

2.4.3 Números Inteiros

Os números inteiros em Python representam todos os números inteiros positivos, negativos e o zero. Eles são armazenados como objetos inteiros e não possuem limite para o tamanho dos números que podem ser armazenados em uma variável inteira. Exemplo de número inteiro em Python:

```
>>> # inteiro positivo
>>> c = 10

>>> # inteiro negativo
>>> r = -200

>>> # zero
>>> v = 0
```

Em geral, uma expressão Python é uma combinação de operadores e operandos. Quando avaliamos uma expressão, obtemos um resultado. Nos próximos exemplos são realizadas operações aritméticas com números inteiros, como adição '+', subtração '-', multiplicação '*', divisão '/', resto da divisão '%' e divisão inteira '//'. Ademais, talvez você esteja mais acostumado com 'x' e '÷' para multiplicação e divisão, mas em Python e quase todas as outras linguagens de programação usam '*' e '/'. A seguir é possível observar o código fonte 2.23 e o resultado 2.24 dos exemplos de número inteiro em Python:

```
>>> # variaveis
>>> v = 14
>>> g = 3
>>>
>>> # soma de inteiros
>>> print(v + g)
17
>>> # subtracao de inteiros
>>> print(v - g)
11
>>> # multiplicacao de inteiros
>>> print(v * g)
42
>>> # divisao de inteiros
>>> print(v / g)
4.666666666666667
>>> # resto da divisao de inteiros
>>> print(v % g)
2
>>> #divisao inteira de inteiros
>>> print(v // g)
4
```


Figura 2.23: Código fonte do exemplo de operações com números inteiros

```

Inteiros.py > ...
1  # variaveis
2  v = 14
3  g = 3
4
5  # soma de inteiros
6  print(v + g)
7
8  # subtracao de inteiros
9  print(v - g)
10
11 # multiplicacao de inteiros
12 print(v * g)
13
14 # divisao de inteiros
15 print(v / g)
16
17 # resto da divisao de inteiros
18 print(v % g)
19
20 #divisao inteira de inteiros
21 print(v // g)

```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 2.24: Resultado do código fonte do exemplo de operações com números inteiros

```

PS C:\Mariana - UENF\PLP\Python\Códigos>
iros.py"
17
11
42
4.666666666666667
2
4

```

Fonte: Criado por Mariana Cossetti Dalfior

2.4.4 Números de Ponto Flutuante

Os números de ponto flutuante são a aproximação do Python do que você chamou de números reais nas aula de matemática. Os números de ponto flutuante são uma aproximação porque, diferentemente dos números reais, os números de ponto flutuante não podem ter um número infinito de dígitos após o ponto decimal. No Python, os números de ponto flutuante são armazenados como objetos float e são aproximados de forma binária, podendo apresentar algumas imprecisões com valores decimais exatos. A seguir é possível observar o código fonte 2.25 e o resultado 2.26 do exemplo de números de ponto flutuante em Python:

```

>>> # variaveis
>>> v = 0.5

```

```
>>> g = 0.2
>>> # soma de floats
>>> f = v + g
>>> print(f)
0.7
```

Figura 2.25: Código fonte do exemplo com números de ponto flutuante

```
Float1.py > ...
1  # variaveis
2  v = 0.5
3  g = 0.2
4
5  # soma de floats
6  f = v + g
7
8  print(f)
```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 2.26: Resultado do código fonte do exemplo com números de ponto flutuante

```
PS C:\Mariana - UENF\PLP\Python\Códigos>
t1.py"
0.7
```

Fonte: Criado por Mariana Cossetti Dalfior

Do mesmo modo que os números inteiros, é possível realizar operações aritméticas com números de ponto flutuante, como adição '+', subtração '-', multiplicação '*', divisão '/', resto da divisão '%' e divisão inteira '//'. A seguir é possível observar o código fonte 2.27 e o resultado 2.28 do exemplo de operações aritméticas com números de ponto flutuante em Python:

```
>>> # variaveis
>>> v = 10.33
>>> g = 5
>>>
>>> # soma de floats
>>> print(v + g)
15.33
>>> # subtracao de floats
>>> print(v - g)
5.33
>>> # multiplicacao de floats
>>> print(v * g)
51.65
>>> # divisao de floats
>>> print(v / g)
2.066
>>> # resto da divisao de floats
```

```
>>> print(v % g)
0.3300000000000007
>>> #divisao inteira de floats
>>> print(v // g)
2.0
```

Figura 2.27: Código fonte do exemplo de operações com números de ponto flutuante

```
Float2.py > ...
1  # variaveis
2  v = 10.33
3  g = 5
4
5  # soma de floats
6  print(v + g)
7
8  # subtracao de floats
9  print(v - g)
10
11 # multiplicacao de floats
12 print(v * g)
13
14 # divisao de floats
15 print(v / g)
16
17 # resto da divisao de floats
18 print(v % g)
19
20 #divisao inteira de floats
21 print(v // g)
```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 2.28: Resultado do código fonte do exemplo de operações com números de ponto flutuante

```
PS C:\Mariana - UENF\PLP\Python\Códigos>
t2.py"
15.33
5.33
51.65
2.066
0.3300000000000007
2.0
```

Fonte: Criado por Mariana Cossetti Dalfior

2.4.5 Números complexos

O último tipo numérico primitivo em Python é o número complexo. Como você pode se recordar, os números complexos têm duas partes: uma parte real e uma parte imaginária. No Python, um

Figura 2.31: Código fonte do exemplo com números booleanos em um *if*

```

Booleanos.py > ...
1  # variaveis
2  v = 6
3  g = 12
4
5  if v > 5 and g > 3:
6      print("{} maior que 5 e {} e maior que 3".format(v, g))

```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 2.32: Resultado do código fonte do exemplo com números booleanos em um *if*

```

PS C:\Mariana - UENF\PLP\Python\Códigos>
Booleanos.py
6 maior que 5 e 12 e maior que 3

```

Fonte: Criado por Mariana Cossetti Dalfior

Ademais, é possível converter qualquer valor ou expressão em um booleano utilizando uma função chamada `bool()`. Essa função retorna `False` caso o valor for considerado "vazio", e `True` caso contrário. A seguir é possível observar o código fonte 2.33 e o resultado 2.34 do exemplo:

```

>>> print(bool(0))
False
>>> print(bool(1898))
True
>>> print(bool(""))
False
>>> print(bool("Vasco"))
True

```

Figura 2.33: Código fonte do exemplo com números booleanos

```

Booleanos2.py
1  print(bool(0))
2
3  print(bool(1898))
4
5  print(bool(""))
6
7  print(bool("Vasco"))

```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 2.34: Resultado do código fonte do exemplo com números booleanos

```
PS C:\Mariana - UENF\PLP\Python\Códigos>
eanos2.py"
False
True
False
True
```

Fonte: Criado por Mariana Cossetti Dalfior

Em resumo, foi evidenciado que o Python oferece suporte a vários tipos diferentes de objetos primitivos: números inteiros, números de ponto flutuante, números complexos, números booleanos, Strings e as operações lógicas.

2.5 Estrutura de Controle e Funções

As estruturas de controle e funções são elementos fundamentais da programação em Python. A seguir serão mostradas algumas dessas estruturas segundo [Gut21]:

2.5.1 O comando *if*

O comando "*if*" em Python é uma estrutura de controle que permite executar determinado bloco de código se uma determinada condição for verdadeira. A sintaxe básica em Python é a seguinte:

```
>>> if condicao:
>>>     # bloco de codigo
```

A seguir é possível observar o código fonte 2.35 e o resultado 2.36 do exemplo com aplicação em Python:

```
>>> # variavel
>>> v = 10
>>>
>>> if v > 5:
>>>     print("Gigante da Colina")
Gigante da Colina
```

Figura 2.35: Código fonte do exemplo do uso do *if*

```

If.py > ...
1  # variavel
2  v = 10
3
4  if v > 5:
5  |     print("Gigante da Colina")
```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 2.36: Resultado do código fonte do exemplo do uso do *if*

```
PS C:\Mariana - UENF\PLP\Python\Códigos>
y"
Gigante da Colina
```

Fonte: Criado por Mariana Cossetti Dalfior

2.5.2 O comando *Else*

O comando 'else' é sempre utilizado em conjunto com o 'if' para fornecer uma alternativa caso a condição do *if* não seja verdadeira. Dessa forma, o bloco de código dentro do *else* só será executado se a condição do *if* for falsa. A sintaxe básica em Python é a seguinte:

```
>>> if condicao:
>>>     # bloco de codigo a ser executado se a condicao for
>>>     # verdadeira
>>> else:
>>>     # bloco de codigo a ser executado se a condicao for
>>>     # falsa
```

A seguir é possível observar o código fonte 2.37 e o resultado 2.38 do exemplo com aplicação em Python:

```
>>> # variavel
>>> v = 12
>>>
>>> if v % 2 == 0:
>>>     print("O numero e par")
>>> else:
>>>     print("O numero e impar")
O numero e par
```

Figura 2.37: Código fonte do exemplo do uso do *else*

```
Else.py > ...
1  # variavel
2  v = 12
3
4  if v % 2 == 0:
5  |     print("O numero e par")
6  else:
7  |     print("O numero e impar")
```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 2.38: Resultado do código fonte do exemplo do uso do *else*

```
PS C:\Mariana - UENF\PLP\Python\Códigos>  
.py"  
O numero e par
```

Fonte: Criado por Mariana Cossetti Dalfior

2.5.3 O comando *Elif*

O comando "*elif*" - abreviação de "*else*" e "*if*" - é utilizado para adicionar uma condição adicional a um bloco *if-else*. Normalmente usado quando existem mais de duas condições possíveis que precisam ser avaliadas. Se a condição do *if* não for atendida, o programa passará para a próxima condição *elif*. Se nenhuma das condições *if* ou *elif* for atendida, o bloco *else* final será executado. A sintaxe do comando *elif* em Python é a seguinte:

```
>>> if condicao1:  
>>>     # Bloco de codigo se condicao1 for verdadeira  
>>> elif condicao2:  
>>>     # Bloco de codigo se condicao2 for verdadeira  
>>> elif condicao3:  
>>>     # Bloco de codigo se condicao3 for verdadeira  
>>> ...  
>>> else:  
>>>     # Bloco de codigo se nenhuma das condicoes anteriores  
>>>     for verdadeira
```

A seguir é possível observar o código fonte 2.39 e o resultado 2.40 do exemplo com aplicação em Python:

```
>>> # variavel  
>>> idade = 20  
>>>  
>>> if idade < 18:  
>>>     print("Voce e menor de idade")  
>>> elif idade >= 18 and idade < 65:  
>>>     print("Voce e adulto")  
>>> else:  
>>>     print("Voce e idoso")  
voce e adulto
```


Figura 2.39: Código fonte do exemplo do uso do *elif*

```
Elif.py > ...
1  # variavel
2  idade = 20
3
4  if idade < 18:
5      print("Voce e menor de idade")
6  elif idade >= 18 and idade < 65:
7      print("Voce e adulto")
8  else:
9      print("Voce e idoso")
```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 2.40: Resultado do código fonte do exemplo do uso do *elif*

```
PS C:\Mariana - UENF\PLP\Python\Códigos>
.py"
Voce e adulto
```

Fonte: Criado por Mariana Cossetti Dalfior

2.5.4 Laço For

O laço `for` é usado para iterar sobre uma sequência, sendo uma estrutura de controle de fluxo que repete um bloco de código um determinado número de vezes, até atingir a condição de parada. Além de que é possível combinar o `for` com outros comandos, como `if`, `else`, `break` e `continue`, para criar lógicas mais complexas dentro do laço. A sintaxe básica do laço `for` em Python é a seguinte:

```
>>> for <variavel> in <sequencia>:
>>>     # bloco de codigo
```

Durante cada iteração do laço `for`, a variável de iteração vai sendo incrementada, e o bloco de código é executado com esse valor. O laço continua até que todos os itens da sequência tenham sido processados. A seguir é possível observar o código fonte 2.41 e o resultado 2.42 do exemplo de um `for` em Python:

```
>>> lista = [1, 2, 3, 4, 5]
>>> for v in lista:
>>>     print(v)
1
2
3
4
5
```

Figura 2.41: Código fonte do exemplo do uso do *for*

```
For.py > ...  
1  lista = [1, 2, 3, 4, 5]  
2  
3  for v in lista:  
4      print(v)
```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 2.42: Resultado do código fonte do exemplo do uso do *for*

```
PS C:\Mariana - UENF\PLP\Python\Códigos>  
py"  
1  
2  
3  
4  
5
```

Fonte: Criado por Mariana Cossetti Dalfior

2.5.5 Laço *While*

O laço de repetição "*while*" é utilizado para repetir um bloco de código até uma condição for falsa. A sua sintaxe geral em Python é a seguinte:

```
>>> while condicao:  
>>> # bloco de codigo a ser executado enquanto a condicao for  
>>> # verdadeira
```

O bloco de código dentro do laço *while* vai ser executado diversas vezes até que a condição seja falsa. É necessário ter bastante cuidado para evitar a criação de um loop infinito, pois a condição nunca se torna falsa e então o programa nunca parará de ser executado. A seguir é possível observar o código fonte 2.43 e o resultado 2.44 do exemplo de um laço *while* em Python:

```
>>> # variaveis  
>>> soma = 0  
>>> v = 1  
>>>  
>>> while v <= 10:  
>>>     soma += v  
>>>     v += 1  
>>> print(soma)  
55
```

Figura 2.43: Código fonte do exemplo do uso do *while*

```
While.py > ...  
1  # variaveis  
2  soma = 0  
3  v = 1  
4  
5  while v <= 10:  
6      soma += v  
7      v += 1  
8  print(soma)
```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 2.44: Resultado do código fonte do exemplo do uso do *while*

```
PS C:\Mariana - UENF\PLP\Python\Códigos>  
e.py"  
55
```

Fonte: Criado por Mariana Cossetti Dalfior



3. Aspectos Avançados da Linguagem Python

Neste capítulo serão apresentados os aspectos avançados da linguagem de programação Python, incluindo classes, dados de coleção e funções geradoras.

3.1 Classes

De acordo com [Ram22] Python oferece algumas maneiras de construir uma classe simples que é apenas uma coleção de campos, com pouca ou nenhuma funcionalidade extra. Esse padrão é conhecido como “classe de dados” — e as classes de dados são um dos pacotes que suportam esse padrão.

3.1.1 Classes base Abstratas

As classes abstratas podem ser utilizadas do módulo *collections.abc* como *Mapping* e *MutableMapping*. Idealmente, uma função deve aceitar argumentos desses tipos abstratos e não tipos concretos. Isso dá mais flexibilidade ao chamador. Considere esta assinatura de função:

```
>>> # Usar abc.Mapping permite que o chamador forneça uma
>>> # instancia de dict, defaultdict, ChainMap, uma subclasse
>>> # UserDict ou qualquer outro tipo que seja um subtipo
>>> # de Mapping.
>>>
>>> from collections.abc import Mapping
>>> def nome2hex(nome: str, color_map: Mapeamento[str, int]) -> str:
```

3.2 Tipos de Dados de Coleção

Os tipos de dados de coleção são utilizados para guardar coleções de valores de acordo com [Per16]. Esses podem ser divididos em dois: tipos sequenciais e tipos conjuntos.

3.2.1 Tipos Sequenciais

Os tipos sequenciais são utilizados para armazenar dados em sequência, ou seja, em uma ordem específica. Existem três tipos de sequência em Python, as listas, tuplas e strings (exibido anteriormente nesse capítulo em 2.4.1):

Listas

As listas são coleções ordenadas de elementos que podem ser de diferentes tipos de dados - como inteiros, strings, booleanos e até outras listas. Elas são uma das estruturas mais utilizadas em Python por serem muito versáteis - podendo ser modificadas removendo, alterando e adicionando novos elementos após a sua criação. As listas são criadas utilizando '[' e seus elementos são separados por vírgulas e indexados de forma numérica - inicia com índice 0 para o primeiro elemento e vai aumentando de um em um para os índices posteriores. A seguir é possível observar o código fonte 3.1 e o resultado 3.2 do exemplo da utilização de listas em Python:

```
>>> # utilizando apenas numeros inteiros
>>> num = [1, 2, 3, 4, 5]
>>> print(num[4])
5

>>> # utilizando apenas strings
>>> vasco = ["Andrey", "Puma", "Pedro Raul"]
>>> print(vasco[1])
Puma

>>> # utilizando de forma mista
>>> junto = [7, "Piton", 5.0, False]
>>> print(junto[3])
False
```

Figura 3.1: Código fonte do exemplo do uso listas

```
Listas1.py > ...
1  # utilizando apenas numeros inteiros
2  num = [1, 2, 3, 4, 5]
3  print(num[4])
4
5  # utilizando apenas strings
6  vasco = ["Andrey", "Puma", "Pedro Raul"]
7  print(vasco[1])
8
9  # utilizando de forma mista
10 junto = [7, "Piton", 5.0, False]
11 print(junto[3])
```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 3.2: Resultado do código fonte do exemplo do uso listas

```
PS C:\Mariana - UENF\PLP\Python\Códigos>  
as1.py"  
5  
Puma  
False
```

Fonte: Criado por Mariana Cossetti Dalfior

É possível realizar operações comuns nas listas utilizando os métodos, como `'.append()'`, `'.extend()'`, `'.insert()'`, `'.remove()'`, `'.pop()'`, `'.index()'`, `'.count()'` e `'.sort()'`. A seguir é possível observar o código fonte 3.3 e o resultado 3.4 do exemplo da utilização de métodos em listas:

```
>>> # utilizando append  
>>> num = [1, 2, 3, 4, 5]  
>>> num.append(6)  
>>> print(num)  
[1, 2, 3, 4, 5, 6]  
  
>>> # utilizando extend  
>>> num = [2, 4, 6]  
>>> impares = [1, 3, 5]  
>>> num.extend(impares)  
>>> print(num)  
[1, 2, 3, 4, 5, 6]  
  
>>> # utilizando insert  
>>> num = [1, 2, 3, 4, 5]  
>>> num.insert(1, 'Vasco')  
>>> print(num)  
[1, 'Vasco', 2, 3, 4, 5]
```

Figura 3.3: Código fonte do exemplo da utilização de métodos em listas

```
Listas2.py > ...
1  # utilizando append
2  num = [1, 2, 3, 4, 5]
3  num.append(6)
4  print(num)
5
6  # utilizando extend
7  num = [2, 4, 6]
8  impares = [1, 3, 5]
9  num.extend(impares)
10 print(num)
11
12 # utilizando insert
13 num = [1, 2, 3, 4, 5]
14 num.insert(1, 'Vasco')
15 print(num)
```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 3.4: Resultado do código fonte do exemplo da utilização de métodos em listas

```
PS C:\Mariana - UENF\PLP\Python\Códigos>
as2.py"
[1, 2, 3, 4, 5, 6]
[2, 4, 6, 1, 3, 5]
[1, 'Vasco', 2, 3, 4, 5]
```

Fonte: Criado por Mariana Cossetti Dalfior

Tuplas

As tuplas são um tipo de dado de coleção, parecidos com as listas, diferente delas possui como características serem imutáveis - não podem ser modificadas depois de sua criação. Além disso, elas geralmente são usadas para o armazenamento de dados que são relacionados, diferente das listas que são utilizadas para armazenar coleções de objetos. As tuplas são criadas de diversas maneiras em Python, porém a mais comum é utilizando parênteses '()'. Exemplo da utilização de tuplas em Python:

```
>>> # criando tupla de valor inteiro
>>> tupla1 = (1, 2, 3, 4, 5)

>>> # criando tupla de string
>>> tupla2 = ('a', 'b', 'c', 'd', 'e')

>>> # criando tupla de booleano
>>> tupla3 = (True, False, True, True)
```

Uma tupla pode conter diferentes tipos de dados, como números, strings e booleanos. As tuplas também podem ser indexadas e fatiadas da mesma maneira que as listas. No entanto, como as tuplas são imutáveis, não é possível alterar um elemento individual da tupla depois que ela foi

criada.

3.2.2 Tipos Conjunto

Na linguagem Python existem dois tipos de conjuntos: o set e o frozenset.

Set

O set é uma coleção de elementos únicos e mutáveis criados usando chaves '{}' ou a função 'set()', onde a ordem dos elementos não é garantida. O conjunto pode ser de diferentes tipos de dados - como inteiros, strings e outras coleções. Ademais, é possível fazer operações matemáticas entre conjuntos - como união, interseção e diferença. A seguir é possível observar o código fonte 3.5 e o resultado 3.6 do exemplo da utilização do set em Python:

```
>>> # Criando um conjunto
>>> conjunto1 = {1, 2, 3, 4, 5}
>>> print(conjunto1)
{1, 2, 3, 4, 5}

>>> # Criando um conjunto vazio
>>> conjunto2 = set()
>>> print(conjunto2)
set()

>>> # Removendo um elemento do conjunto
>>> conjunto1.remove(3)
>>> print(conjunto1)
{1, 2, 4, 5}

>>> # Adicionando um elemento ao conjunto
>>> conjunto1.add(6)
>>> print(conjunto1)
{1, 2, 3, 4, 5, 6}

>>> # Operacoes entre conjuntos
>>> conjunto2 = {4, 5, 6, 7, 8}
>>> print(conjunto1.union(conjunto2))
{1, 2, 3, 4, 5, 6, 7, 8}
>>> print(conjunto1.intersection(conjunto2))
{4, 5}
>>> print(conjunto1.difference(conjunto2))
{1, 2, 3}
```

Figura 3.5: Código fonte do exemplo do uso do set

```

Set.py > ...
1  # Criando um conjunto
2  conjunto1 = {1, 2, 3, 4, 5}
3  print(conjunto1)
4
5  # Criando um conjunto vazio
6  conjunto2 = set()
7  print(conjunto2)
8
9  # Removendo um elemento do conjunto
10 conjunto1.remove(3)
11 print(conjunto1)
12 {1, 2, 3, 4, 5}
13
14 # Adicionando um elemento ao conjunto
15 conjunto1.add(6)
16 print(conjunto1)
17 {1, 2, 3, 4, 5, 6}
18
19 # Operacoes entre conjuntos
20 conjunto2 = {4, 5, 6, 7, 8}
21 print(conjunto1.union(conjunto2))
22 print(conjunto1.intersection(conjunto2))
23 print(conjunto1.difference(conjunto2))

```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 3.6: Resultado do código fonte do exemplo do uso do set

```

PS C:\Mariana - UENF\PLP\Python\Códigos>
py"
{1, 2, 3, 4, 5}
set()
{1, 2, 4, 5}
{1, 2, 4, 5, 6}
{1, 2, 4, 5, 6, 7, 8}
{4, 5, 6}
{1, 2}

```

Fonte: Criado por Mariana Cossetti Dalfior

Frozenset

O frozenset é uma coleção de elementos únicos e imutáveis criados usando a função `'frozenset()'`, porém a ordem desses elementos não é garantida. O conjunto imutável pode ser de diferentes tipos de dados - como inteiros, strings e outras coleções. Contudo, não possibilita a adição, remoção ou mudança de elementos após a criação. A seguir é possível observar o código fonte 3.7 e o resultado 3.8 do exemplo da utilização do frozenset em Python:

```
>>> # Criando um conjunto imutavel
>>> num = frozenset([2, 4, 6, 8, 10])
>>> print(num)
frozenset({2, 4, 6, 8, 10})
```

Figura 3.7: Código fonte do exemplo do uso do *frozenset*

```
Frozenset.py > ...
1 # Criando um conjunto imutavel
2 num = frozenset([2, 4, 6, 8, 10])
3 print(num)
```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 3.8: Resultado do código fonte do exemplo do uso do *frozenset*

```
PS C:\Mariana - UENF\PLP\Python\Códigos>
enst.py"
frozenset({2, 4, 6, 8, 10})
```

Fonte: Criado por Mariana Cossetti Dalfior

3.2.3 Tipos Mapeamento

Os tipo de dados de mapeamento se referem a uma coleção de pares chave-valor, em que os valores podem ser acessados através de suas chaves correspondentes. Os dois tipos principais de mapeamento em Python são:

Dicionários

Os dicionário em Python são um tipo de mapeamento que possibilitam armazenar pares com valores em suas chaves correspondentes. Diferente dos valores que podem ser alterados e de qualquer tipo de dados, as chaves são únicas e imutáveis. A formatação utilizada nos dicionário são as '{}' e os pares de chave-valor são separados por vírgulas. A seguir é possível observar o código fonte 3.9 e o resultado 3.10 do exemplo da utilização de um dicionário em Python:

```
>>> # definindo um dicionario
>>> time_vasco = {"nome": "Capasso", "idade": 27, "posicao":
    "zagueiro"}
>>>
>>> # acessando valores no dicionario
>>> print(time_vasco["nome"])
Capasso
>>>
>>> # atualizando valores
>>> time_vasco["nome"] = "Robson Bambu"
>>> time_vasco["idade"] = 25
>>>
>>> # acessando valores atualizados no dicionario
>>> print(time_vasco["nome"])
Robson Bambu
```

Figura 3.9: Código fonte do exemplo do uso de dicionário

```
Dicionario.py > ...
1  # definindo um dicionario
2  time_vasco = {"nome": "Capasso", "idade": 27, "posicao": "zagueiro"}
3
4  # acessando valores no dicionario
5  print(time_vasco["nome"])
6
7  # atualizando valores
8  time_vasco["nome"] = "Robson Bambu"
9  time_vasco["idade"] = 25
10
11 # acessando valores atualizados no dicionario
12 print(time_vasco["nome"])
```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 3.10: Resultado do código fonte do exemplo do uso de dicionário

```
PS C:\Mariana - UENF\PLP\Python\Códigos>
onario.py"
Capasso
Robson Bambu
```

Fonte: Criado por Mariana Cossetti Dalfior

Além de poder modificar os valores contidos nas chaves também é possível usar métodos como 'keys()' e 'values()' para conseguir uma lista com todas as chaves ou valores em um dicionário. A seguir é possível observar o código fonte 3.11 e o resultado 3.12 do exemplo da utilização desses métodos:

```
>>> # imprime uma lista de chaves
>>> print(time_vasco.keys())
>>>
>>> # imprime uma lista de valores
>>> print(time_vasco.values())
dict_keys(['nome', 'idade', 'posicao'])
dict_keys(['Robson Bambu', 25, 'zagueiro'])
```

Figura 3.11: Código fonte do exemplo da utilização de métodos em dicionários

```
14 # imprime uma lista de chaves
15 print(time_vasco.keys())
16
17 # imprime uma lista de valores
18 print(time_vasco.values())
```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 3.12: Resultado do código fonte do exemplo da utilização de métodos em dicionários

```
PS C:\Mariana - UENF\PLP\Python\Códigos> & C:/l  
onario.py"  
dict_keys(['nome', 'idade', 'posicao'])  
dict_values(['Robson Bambu', 25, 'zagueiro'])
```

Fonte: Criado por Mariana Cossetti Dalfior

Defaultdicts

Os defaultdict são objetos parecidos com os dicionários, porém possuem como diferencial a possibilidade de definir valores padrão para chaves que não existem. Mesmo com isso, eles agem como um dicionário comum, mas quando uma chave é acessada e ela ainda não existe, no lugar de levantar um erro chamado 'KeyError', ele retorna um valor padrão que foi definido pelo usuário. Esse valor é determinado na criação do defaultdict. A seguir é possível observar o código fonte 3.13 e o resultado 3.14 do exemplo que retornará '[]' quando a chave que não existe for acessada:

```
>>> #importando biblioteca  
>>> from collections import defaultdict  
>>>  
>>> # criando um defaultdict com uma lista vazia  
>>> vasco = defaultdict(list)  
>>>  
>>> # adicionando valores ao defaultdict  
>>> vasco['goleiro'].append('Leo Jardim')  
>>> vasco['goleiro'].append('Ivan')  
>>> vasco['atacante'].append('Gabriel Pec')  
>>> vasco['atacante'].append('Pedro Raul')  
>>>  
>>> # acessando valores no defaultdict  
>>> print(vasco['goleiro'])  
['Leo Jardim', 'Ivan']  
>>> print(vasco['atacante'])  
['Gabriel Pec', 'Pedro Raul']  
>>> print(vasco['zagueiro'])  
[]
```

Figura 3.13: Código fonte do exemplo do uso do *defaultdict*

```

Defaultdict.py > ...
1  # importando biblioteca
2  from collections import defaultdict
3
4  # criando um defaultdict com uma lista vazia
5  vasco = defaultdict(list)
6
7  # adicionando valores ao defaultdict
8  vasco['goleiro'].append('Leo Jardim')
9  vasco['goleiro'].append('Ivan')
10 vasco['atacante'].append('Gabriel Pec')
11 vasco['atacante'].append('Pedro Raul')
12
13 # acessando valores no defaultdict
14 print(vasco['goleiro'])
15 print(vasco['atacante'])
16 print(vasco['zagueiro'])

```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 3.14: Resultado do código fonte do exemplo do uso do *defaultdict*

```

PS C:\Mariana - UENF\PLP\Python\Códigos>
ultdict.py"
['Leo Jardim', 'Ivan']
['Gabriel Pec', 'Pedro Raul']
[]

```

Fonte: Criado por Mariana Cossetti Dalfior

3.2.4 Metaprogramação de classes

A metaprogramação de classe é a arte de criar ou customizar classes em tempo de execução. As classes são objetos de primeira classe em Python, portanto, uma função pode ser usada para criar uma nova classe a qualquer momento, sem usar a palavra-chave *class*. Decoradores de classe também são funções, mas projetados para inspecionar, alterar e até mesmo substituir a classe decorada por outra classe. Finalmente, as metaclasses são a ferramenta mais avançada para a metaprogramação de classes: elas permitem que você crie categorias totalmente novas de classes com características especiais.

Decoradores de classe

Um decorador de classe é um recurso de chamada que se comporta de forma semelhante a um decorador de função: ele obtém a classe decorada como argumento e deve retornar uma classe para substituir a classe decorada. Os decoradores de classe geralmente retornam a própria classe decorada, depois de injetar mais métodos nela por meio da atribuição de atributos. Provavelmente, a razão mais comum para escolher um decorador de classe em vez do simples `__init_subclass__` é evitar a interferência com outros recursos de classe, como herança e metaclasses. A seguir é possível observar o código fonte 3.15 e o resultado 3.16 do exemplo de decoradores:

```
>>> @checked
```

```

>>> @dataclass
>>> class Filme:
>>>     titulo: str
>>>     ano: int
>>>     preco: float
>>>
>>> filme = Filme(titulo= 'O Poderoso Chefao', ano= 1972, preco= 137.6)
>>>
>>> print(filme.titulo)
O Poderoso Chefao
>>> print(filme)
Filme(titulo= 'O Poderoso Chefao', ano= 1972, preco= 137.6)

```

Figura 3.15: Código fonte do exemplo do uso de decoradores de classe

Decoradores.py > ...

```

18 @checked
19 @dataclass
20 class Filme:
21     titulo: str
22     ano: int
23     preco: float
24
25 filme = Filme(titulo= 'O Poderoso Chefao', ano= 1972, preco= 137.6)
26
27 print(filme.titulo)
28 print(filme)

```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 3.16: Resultado do código fonte do exemplo do uso de decoradores de classe

```

PS C:\Mariana - UENF\PLP\Python\Códigos> & C:/Users/mari
radores.py"
O Poderoso Chefao
Filme(titulo='O Poderoso Chefao', ano=1972, preco=137.6)

```

Fonte: Criado por Mariana Cossetti Dalfior

Metaclasses

A classe de tipo é uma metaclasses: uma classe que constrói classes. Em outras palavras, as instâncias da classe de tipo são classes. A biblioteca padrão fornece algumas outras metaclasses, mas `type` é a padrão. A seguir é possível observar o código fonte 3.17 e o resultado 3.18 desse exemplo:

```

>>> print(type(7))
<class 'int'>
>>> print(type(int))
<class 'type'>
>>>

```

Figura 3.17: Código fonte do exemplo do uso de metaclasses

```

❏ Metaclasses.py
1  print(type(7))
2
3  print(type(int))

```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 3.18: Resultado do código fonte do exemplo do uso de metaclasses

```

PS C:\Mariana - UENF\PLP\Python\Códigos>
classe.py"
<class 'int'>
<class 'type'>

```

Fonte: Criado por Mariana Cossetti Dalfior

3.2.5 Heranças

A herança é um mecanismo da programação orientada a objetos que visa facilitar a reutilização de código. A ideia fundamental é que as classes possam ser organizadas em uma hierarquia, onde as novas classes podem herdar os métodos e propriedades das classes existentes (que podem ter sido herdadas de outras classes anteriores) e, ao mesmo tempo, adicionar seus próprios métodos e propriedades.

Herança Simples

Na herança, é comum utilizar a chamada "herança simples", em que as novas classes são derivadas das classes existentes. É possível criar múltiplas classes derivadas, formando uma hierarquia de classes. Na busca por métodos e propriedades, a procura ocorre de baixo para cima na hierarquia, de forma similar à busca em *namespaces* locais e globais. A seguir é possível observar o código fonte 3.19 e o resultado 3.20 do exemplo de herança simples:

```

>>> class Pendrive(object):
>>> def __init__(self, tamanho, interface='2.0'):
>>> self.tamanho = tamanho
>>> self.interface = interface
>>>
>>> class MP3Player(Pendrive):
>>> def __init__(self, tamanho, interface='2.0', turner=False):
>>> super().__init__(tamanho, interface)
>>> self.turner = turner
>>>
>>> mp3 = MP3Player(1024)
>>> print('%s\n%s\n%s' % (mp3.tamanho, mp3.interface, mp3.turner))
1024
2.0
False

```


Figura 3.19: Código fonte do exemplo do uso de herança simples

```

HerancaSimples.py > ...
1  class Pendrive(object):
2      def __init__(self, tamanho, interface='2.0'):
3          self.tamanho = tamanho
4          self.interface = interface
5
6  class MP3Player(Pendrive):
7      def __init__(self, tamanho, interface='2.0', turner=False):
8          super().__init__(tamanho, interface)
9          self.turner = turner
10
11  mp3 = MP3Player(1024)
12  print('%s\n%s\n%s' % (mp3.tamanho, mp3.interface, mp3.turner))

```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 3.20: Resultado do código fonte do exemplo do uso de herança simples

```

PS C:\Mariana - UENF\PLP\Python\Códigos>
ncaSimples.py"
1024
2.0
False

```

Fonte: Criado por Mariana Cossetti Dalfior

Essa abordagem permite que as classes filhas herdem o comportamento e as características das classes pai, enquanto têm a flexibilidade de adicionar ou modificar essas características conforme necessário, dentro da nova classe. A herança, portanto, promove a reutilização de código ao facilitar o aproveitamento de implementações existentes e a criação de classes especializadas com base nessas implementações.

Heranças Múltiplas

A herança múltipla permite que uma nova classe seja derivada de duas ou mais classes existentes ao mesmo tempo. A seguir é possível observar o código fonte 3.21 e o resultado 3.22 do exemplo de herança múltipla:

```

>>> class Terrestre(object):
>>>     se_move_em_terra = True
>>>
>>>     def __init__(self, velocidade=100):
>>>         self.velocidade_em_terra = velocidade
>>>
>>> class Aquatico(object):
>>>     se_move_na_agua = True
>>>
>>>     def __init__(self, velocidade=5):
>>>         self.velocidade_agua = velocidade
>>>
>>> class Carro(Terrestre):

```

```
>>>     rodas = 4
>>>
>>>     def __init__(self, velocidade=120, pistoes=4):
>>>         self.pistoes = pistoes
>>>         super().__init__(velocidade=velocidade)
>>>
>>> class Barco(Aquatico):
>>>
>>>     def __init__(self, velocidade=6, helices=1):
>>>         self.helices = helices
>>>         super().__init__(velocidade=velocidade)
>>>
>>> class Anfibio(Carro, Barco):
>>>
>>>     def __init__(self, velocidade_em_terra=80,
>>>                     velocidade_na_agua=4, pistoes=6, helices=2):
>>>         Carro.__init__(self, velocidade=velocidade_em_terra,
>>>                         pistoes=pistoes)
>>>         Barco.__init__(self, velocidade=velocidade_na_agua,
>>>                         helices=helices)
>>>
>>> novo_anfibio = Anfibio()
>>> for atr in dir(novo_anfibio):
>>>     if not atr.startswith('__'):
>>>         print(atr, '=', getattr(novo_anfibio, atr))
helices = 2
pistoes = 6
rodas = 4
se_move_em_terra = True
se_move_em_agua = True
velocidade_agua = 4
velocidade_terra = 80
```

Figura 3.21: Código fonte do exemplo do uso de herança múltipla

```

HerancaMultipla.py > ...
1 class Terrestre(object):
2     se_move_em_terra = True
3
4     def __init__(self, velocidade=100):
5         self.velocidade_em_terra = velocidade
6
7 class Aquatico(object):
8     se_move_na_agua = True
9
10    def __init__(self, velocidade=5):
11        self.velocidade_agua = velocidade
12
13    class Carro(Terrestre):
14        rodas = 4
15
16        def __init__(self, velocidade=120, pistoes=4):
17            self.pistoes = pistoes
18            super().__init__(velocidade=velocidade)
19
20    class Barco(Aquatico):
21
22        def __init__(self, velocidade=6, helices=1):
23            self.helices = helices
24            super().__init__(velocidade=velocidade)
25
26    class Anfibio(Carro, Barco):
27
28        def __init__(self, velocidade_em_terra=80, velocidade_na_agua=4, pistoes=6, helices=2):
29            Carro.__init__(self, velocidade=velocidade_em_terra, pistoes=pistoes)
30            Barco.__init__(self, velocidade=velocidade_na_agua, helices=helices)
31
32    novo_anfibio = Anfibio()
33    for atr in dir(novo_anfibio):
34        if not atr.startswith('__'):
35            print(atr, '=', getattr(novo_anfibio, atr))

```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 3.22: Resultado do código fonte do exemplo do uso de herança múltipla

```

PS C:\Mariana - UENF\PLP\Python\Códigos>
ncaMultipla.py"
helices = 2
pistoes = 6
rodas = 4
se_move_em_terra = True
se_move_na_agua = True
velocidade_agua = 4
velocidade_em_terra = 80

```

Fonte: Criado por Mariana Cossetti Dalfior

A diferença mais notável em relação à herança simples está na ordem de resolução de métodos (MRO, do inglês Method Resolution Order). A herança múltipla é um recurso controverso devido à sua complexidade, podendo tornar o design do código confuso e menos claro.

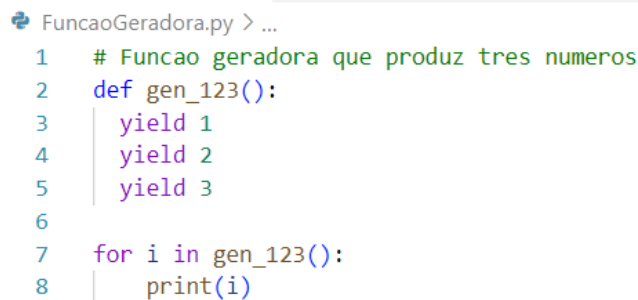
3.3 Funções Geradoras

Segundo [SKSP21], a única sintaxe que distingue uma função simples de uma geradora é o fato de que a última possui uma palavra-chave *yield* em algum lugar de seu corpo. Qualquer função

Python que tenha essa palavra-chave em seu corpo é uma função geradora: uma função que, quando chamada, retorna um objeto gerador. Em outras palavras, uma função de geradora é uma fábrica de geradores. A seguir é possível observar o código fonte 3.21 e o resultado 3.22 do exemplo utilizando função geradora:

```
>>> # Funcao geradora que produz tres numeros
>>> def gen_123():
>>>     yield 1
>>>     yield 2
>>>     yield 3
>>>
>>> for i in gen_123():
>>>     print(i)
1
2
3
```

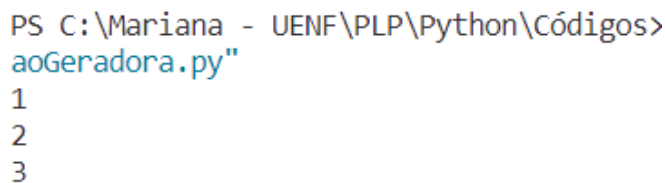
Figura 3.23: Código fonte do exemplo do uso de funções geradoras



```
FuncaoGeradora.py > ...
1  # Funcao geradora que produz tres numeros
2  def gen_123():
3      yield 1
4      yield 2
5      yield 3
6
7  for i in gen_123():
8      print(i)
```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 3.24: Resultado do código fonte do exemplo do uso de funções geradoras



```
PS C:\Mariana - UENF\PLP\Python\Códigos>
aoGeradora.py"
1
2
3
```

Fonte: Criado por Mariana Cossetti Dalfior

Dessa forma, foi criada uma função geradora `gen_123()` e para ela foram criadas três instruções `yield` para retornar os valores 1, 2 e 3, respectivamente. Depois foi criado um loop com o `for` para que esses números fossem mostrados, através do `print(i)`.



4. Aplicações da Linguagem Python

Como foi observado nos capítulos anteriores a linguagem Python pode ser aplicada de diversas maneiras. A seguir será apresentado algumas dessas aplicações.

4.1 Operações básicas

Em Python é possível realizar as operações básicas de forma rápida e prática. As mais conhecidas são adição, subtração, multiplicação e divisão. Elas são representadas da seguinte maneira: [Ban18]

Operações	Operador
Adição	+
Subtração	-
Multiplicação	*
Divisão	/
Divisão inteira	//
Resto da divisão (módulo)	%
Potenciação	**

Tabela 4.1: Operações e seus operadores em Python

Essa aplicação é praticamente uma calculadora, em que os objetos *x* e *y* entram no código com o *input* e as operações ocorrem através das funções pré-definidas. Essas são atribuídas a variável resultado e por fim é mostrado na tela com o *print*. A seguir é possível observar o código fonte 4.1 e o resultado 4.2 da sua aplicação:

```
>>> def somar(a, b):
>>>     return a + b
>>>
>>> def subtrair(a, b):
```

```
>>> return a - b
>>>
>>> def multiplicar(a, b):
>>> return a * b
>>>
>>> def dividir(a, b):
>>> if b != 0:
>>> return a / b
>>> else:
>>> return "Erro: divisao por zero"
>>>
>>> print("Selecione a operacao:")
>>> print("1. Somar")
>>> print("2. Subtrair")
>>> print("3. Multiplicar")
>>> print("4. Dividir")
>>>
>>> operacao = input("Qual operacao? (1,2,3,4): ")
>>> x = float(input("Digite o primeiro numero: "))
>>> y = float(input("Digite o segundo numero: "))
>>>
>>> if operacao == '1':
>>> resultado = somar(x, y)
>>> print("Resultado:", resultado)
>>> elif operacao == '2':
>>> resultado = subtrair(x, y)
>>> print("Resultado:", resultado)
>>> elif operacao == '3':
>>> resultado = multiplicar(x, y)
>>> print("Resultado:", resultado)
>>> elif operacao == '4':
>>> resultado = dividir(x, y)
>>> print("Resultado:", resultado)
>>> else:
>>> print("Escolha invalida")
>>>
```

Figura 4.1: Código fonte da calculadora

```

Calculadora.py > ...
1  def somar(a, b):
2      return a + b
3
4  def subtrair(a, b):
5      return a - b
6
7  def multiplicar(a, b):
8      return a * b
9
10 def dividir(a, b):
11     if b != 0:
12         return a / b
13     else:
14         return "Erro: divisão por zero"
15
16 print("Selecione a operação:")
17 print("1. Somar")
18 print("2. Subtrair")
19 print("3. Multiplicar")
20 print("4. Dividir")
21
22 operacao = input("Qual operacao? (1,2,3,4): ")
23 x = float(input("Digite o primeiro número: "))
24 y = float(input("Digite o segundo número: "))
25
26 if operacao == '1':
27     resultado = somar(x, y)
28     print("Resultado:", resultado)
29 elif operacao == '2':
30     resultado = subtrair(x, y)
31     print("Resultado:", resultado)
32 elif operacao == '3':
33     resultado = multiplicar(x, y)
34     print("Resultado:", resultado)
35 elif operacao == '4':
36     resultado = dividir(x, y)
37     print("Resultado:", resultado)
38 else:
39     print("Escolha inválida")

```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 4.2: Resultado do código fonte da calculadora

```

PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

O Windows PowerShell
Copyright (C) Microsoft Corporation. Todos os

Instale o PowerShell mais recente para obter r
1. Somar
2. Subtrair
3. Multiplicar
4. Dividir
Qual operacao? (1,2,3,4): 1
Digite o primeiro número: 6
Digite o segundo número: 70
Resultado: 76.0
PS C:\Mariana - UENF\PLP\Python\Códigos>

```

Fonte: Criado por Mariana Cossetti Dalfior

Além dessas operações mostradas acima, existe a radiciação. Que possui uma função já definida capaz de ser importada do módulo *math*. O código fonte 4.3 dessa operação e seu resultado 4.4 serão apresentados a seguir:

```
>>> import math
>>>
>>> raiz = float(input("\nValor para descobrir a raiz quadrada:
"))
>>> resultado = math.sqrt(raiz)
>>>
>>> print("\nO valor da raiz quadrada é: ", resultado)
```

Figura 4.3: Código fonte da radiciação

```
Radiciação.py > ...
1  import math
2
3  raiz = float(input("\nValor para descobrir a raiz quadrada: "))
4  resultado = math.sqrt(raiz)
5
6  print("\nO valor da raiz quadrada é: ", resultado)
```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 4.4: Resultado do código fonte da radiciação

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE
PS C:\Mariana - UENF\PLP\Python\Códigos> &
Valor para descobrir a raiz quadrada: 49
O valor da raiz quadrada é:
7.0
PS C:\Mariana - UENF\PLP\Python\Códigos> □
```

Fonte: Criado por Mariana Cossetti Dalfior

4.2 Programas com Objetos

Para manipular dados em Python, é comum criar objetos que representem esses dados e facilitem sua manipulação. Existem diversas formas para a criação de objetos e também diversas maneiras como, utilizando listas, matrizes, tuplas, dicionários, Pandas DataFrames, etc. Nessa aplicação foi criada duas classes e duas funções para realizar a instanciação dessas classes, os quais recebem dois parâmetros responsáveis pela definição de atributos para os objetos. A seguir é possível observar o código fonte 4.5 e o resultado 4.6 da sua aplicação:

```
>>> class Gato:
>>> def __init__(self, nome, idade):
>>> self.nome = nome
>>> self.idade = idade
>>>
>>> class Dono:
>>> def __init__(self, nome, profissao):
>>> self.nome = nome
```



```
>>> self.profissao = profissao
>>>
>>> gato = Gato("Auau", 1)
>>> dono = Dono("Mariana", "Estudante")
>>>
>>> print("\nInformacoes do Cachorro:")
>>> print("Nome:", gato.nome)
>>> print("Idade:", gato.idade)
>>> print("\nInformacoes do Dono:")
>>> print("Nome:", dono.nome)
>>> print("Profissao:", dono.profissao)
```

Figura 4.5: Código fonte de Programa com Objetos

```
Objetos.py > ...
1 class Gato:
2     def __init__(self, nome, idade):
3         self.nome = nome
4         self.idade = idade
5
6 class Dono:
7     def __init__(self, nome, profissao):
8         self.nome = nome
9         self.profissao = profissao
10
11 gato = Gato("Auau", 1)
12 dono = Dono("Mariana", "Estudante")
13
14 print("\nInformações do Cachorro:")
15 print("Nome:", gato.nome)
16 print("Idade:", gato.idade)
17 print("\nInformações do Dono:")
18 print("Nome:", dono.nome)
19 print("Profissão:", dono.profissao)
```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 4.6: Resultado do código fonte de Programa com Objetos

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

PS C:\Mariana - UENF\PLP\Python\Códigos> & (

Informações do Cachorro:
Nome: Auau
Idade: 1

Informações do Dono:
Nome: Mariana
Profissão: Estudante
PS C:\Mariana - UENF\PLP\Python\Códigos> █
```

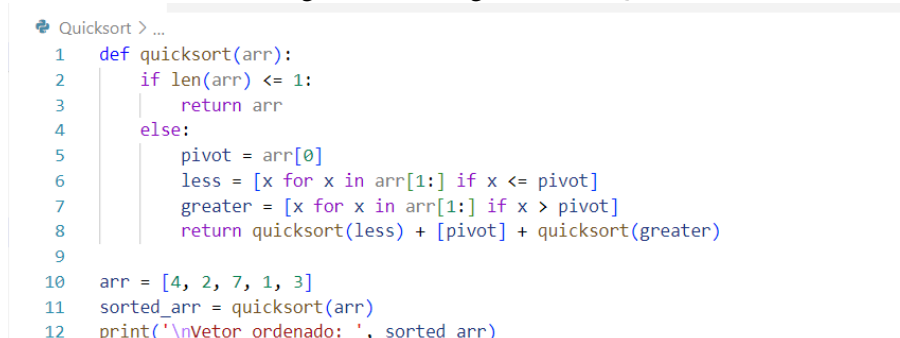
Fonte: Criado por Mariana Cossetti Dalfior

4.3 O algoritmo Quicksort

O Quicksort é um algoritmo de ordenação com base no método de divisão e conquista, na etapa da divisão é onde tudo realmente acontece. Na aplicação a seguir há a ordenação de um vetor em ordem crescente, para isso o algoritmo começa comparando todos os valores com o primeiro elemento. Após isso há a criação de um subvetor para elementos menores que o primeiro e um outro subvetor para elementos maiores, ordenando esses novamente utilizando o quicksort na recursividade, e em seguida é inserido o primeiro elemento no seu devido lugar. A seguir é possível observar o código fonte 4.8 da sua aplicação, retirados do livro [Ram22]:

```
>>> def quicksort(arr):
>>> if len(arr) <= 1:
>>> return arr
>>> else:
>>> pivot = arr[0]
>>> less = [x for x in arr[1:] if x <= pivot]
>>> greater = [x for x in arr[1:] if x > pivot]
>>> return quicksort(less) + [pivot] + quicksort(greater)
>>>
>>> arr = [4, 2, 7, 1, 3]
>>> sorted_arr = quicksort(arr)
>>> print('\nVetor ordenado: ', sorted_arr)
```

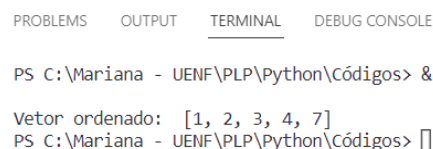
Figura 4.7: Código fonte de Quicksort



```
Quicksort > ...
1 def quicksort(arr):
2     if len(arr) <= 1:
3         return arr
4     else:
5         pivot = arr[0]
6         less = [x for x in arr[1:] if x <= pivot]
7         greater = [x for x in arr[1:] if x > pivot]
8         return quicksort(less) + [pivot] + quicksort(greater)
9
10 arr = [4, 2, 7, 1, 3]
11 sorted_arr = quicksort(arr)
12 print('\nVetor ordenado: ', sorted_arr)
```

Fonte: Exemplo 5.2

Figura 4.8: Resultado do código fonte de Quicksort



```
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

PS C:\Mariana - UENF\PLP\Python\Códigos> &
Vetor ordenado: [1, 2, 3, 4, 7]
PS C:\Mariana - UENF\PLP\Python\Códigos> []
```

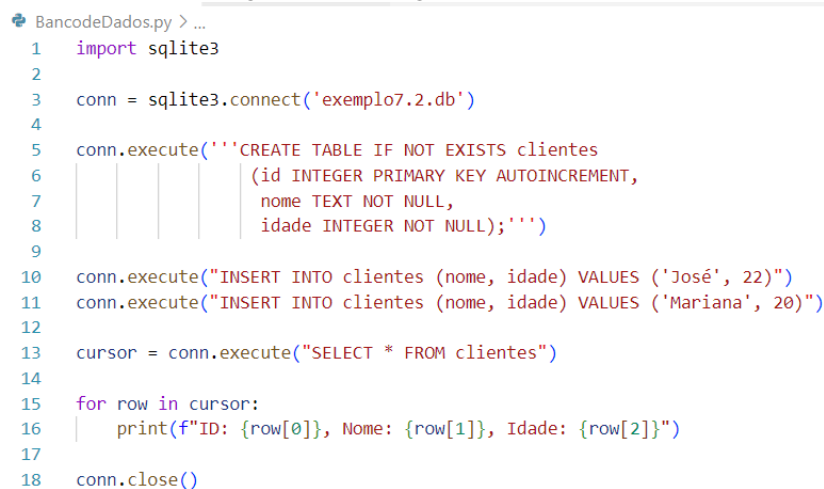
Fonte: Exemplo 5.2

4.4 Aplicações com Banco de Dados

A linguagem Python é amplamente utilizada para manipular bancos de dados e facilitar o estudo e análise desses dados. Python oferece várias maneiras de trabalhar com bancos de dados e oferece uma ampla gama de opções para cientistas de dados. Uma maneira é utilizar uma biblioteca padrão chamada "sqlite3" que permite criar e manipular bancos de dados *SQLite*. Que é um banco de dados leve e amplamente utilizado, ideal para projetos menores ou aplicações que não exijam uma infraestrutura mais complexa. Nessa aplicação, é criada uma conexão com um banco de dados, em seguida cria uma tabela e insere dados nela, após isso esses dados são recuperados e impressos com o *print*. A seguir é possível observar o código fonte 4.9 e o resultado 4.10 da sua aplicação, retirados do livro [Gut21]:

```
>>> import sqlite3
>>>
>>> conn = sqlite3.connect('exemplo7.2.db')
>>>
>>> conn.execute('''CREATE TABLE IF NOT EXISTS clientes
>>> (id INTEGER PRIMARY KEY AUTOINCREMENT,
>>> nome TEXT NOT NULL,
>>> idade INTEGER NOT NULL);''')
>>>
>>> conn.execute("INSERT INTO clientes (nome, idade) VALUES
('Jose', 22)")
>>> conn.execute("INSERT INTO clientes (nome, idade) VALUES
('Mariana', 20)")
>>>
>>> cursor = conn.execute("SELECT * FROM clientes")
>>>
>>> for row in cursor:
>>> print(f"ID: {row[0]}, Nome: {row[1]}, Idade: {row[2]}")
>>>
>>> conn.close()
```

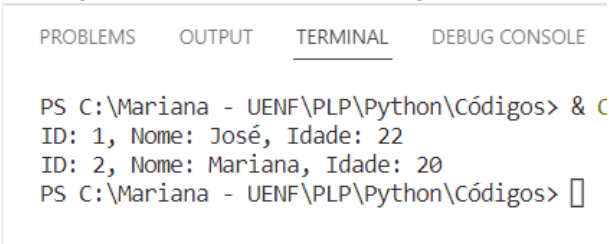
Figura 4.9: Código fonte de banco de dados



```
BancodeDados.py > ...
1  import sqlite3
2
3  conn = sqlite3.connect('exemplo7.2.db')
4
5  conn.execute('''CREATE TABLE IF NOT EXISTS clientes
6  |               | (id INTEGER PRIMARY KEY AUTOINCREMENT,
7  |               | nome TEXT NOT NULL,
8  |               | idade INTEGER NOT NULL);''')
9
10 conn.execute("INSERT INTO clientes (nome, idade) VALUES ('José', 22)")
11 conn.execute("INSERT INTO clientes (nome, idade) VALUES ('Mariana', 20)")
12
13 cursor = conn.execute("SELECT * FROM clientes")
14
15 for row in cursor:
16 |     print(f"ID: {row[0]}, Nome: {row[1]}, Idade: {row[2]}")
17
18 conn.close()
```

Fonte: Exemplo 7.2

Figura 4.10: Resultado do código fonte de banco de dados



```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

PS C:\Mariana - UENF\PLP\Python\Códigos> & c
ID: 1, Nome: José, Idade: 22
ID: 2, Nome: Mariana, Idade: 20
PS C:\Mariana - UENF\PLP\Python\Códigos> □
```

Fonte: Exemplo 7.2

4.5 Programa de Cálculo Numérico

Em Python, o Cálculo Numérico é um campo complexo e possui diversas bibliotecas que oferecem suporte à manipulação e análise de dados numéricos. É notável nessas bibliotecas o *NumPy*, que fornece forte suporte para muitos programas e funções matemáticas, e também *SciPy*, que fornece recursos adicionais como a integração, interpolação e otimização numérica. Nessa aplicação para encontrar a raiz da função é aplicado de forma repetida o método de bisseção, o qual vai atualizando o intervalo e verificando o critério de parada. A seguir é possível observar o código fonte 4.11 e o resultado 4.12 da sua aplicação, retirados do livro [?]:

```
>>> def calcula_f(x):
>>> return x ** 3 + 5 * x ** 2 - 5 * x - 12
>>>
>>> iteracao_maxima = 100
>>>
>>> iteracao = 0
>>>
>>> erro_definido = 0.0001
>>>
>>> parar = 0
>>>
>>> a = -6.0
>>> b = -4.0
>>>
>>> y1 = calcula_f(a)
>>> y2 = calcula_f(b)
>>>
>>> x0_anterior = 2*b
>>>
>>> if y1 * y2 > 0:
>>> print (u'erro de Execucao - Redefinir valores de a e b.')
>>> else:
>>> while parar == 0:
>>> iteracao = iteracao + 1
>>> x0 = (a + b) / 2
>>> y0 = calcula_f(x0)
>>>
>>> if y1 * y0 > 0:
>>> a = x0
```

```
>>>
>>> if y2*y0 > 0:
>>> b = x0
>>>
>>> erro = abs(x0 - x0_anterior)
>>>
>>> x0_anterior = x0
>>>
>>> if (iteracao > iteracao_maxima) or (erro < erro_definido):
>>> parar = 1
>>>
>>> print (u'\nTotal de Iteracoes')
>>> print (iteracao)
>>>
>>> print ('\nValor de x')
>>> print ('x = ', x0)
>>>
>>> print ('\nerro encontrado')
>>> print ('erro = ', erro)
```

Figura 4.11: Código fonte de Cálculo Numérico

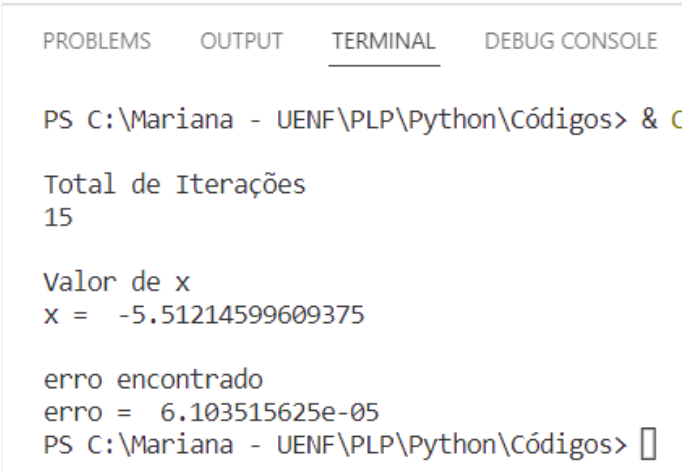
```

CalculoNumerico.py > ...
1  def calcula_f(x):
2      return x ** 3 + 5 * x ** 2 - 5 * x - 12
3
4  iteracao_maxima = 100
5
6  iteracao = 0
7
8  erro_definido = 0.0001
9
10 parar = 0
11
12 a = -6.0
13 b = -4.0
14
15 y1 = calcula_f(a)
16 y2 = calcula_f(b)
17
18 x0_anterior = 2*b
19
20 if y1 * y2 > 0:
21     print (u'erro de Execução - Redefinir valores de a e b.')
22 else:
23     while parar == 0:
24         iteracao = iteracao + 1
25         x0 = (a + b) / 2
26         y0 = calcula_f(x0)
27
28         if y1 * y0 > 0:
29             a = x0
30
31         if y2*y0 > 0:
32             b = x0
33
34         erro = abs(x0 - x0_anterior)
35
36         x0_anterior = x0
37
38         if (iteracao > iteracao_maxima) or (erro < erro_definido):
39             parar = 1
40
41     print (u'\nTotal de Iterações')
42     print (iteracao)
43
44     print ('\nValor de x')
45     print ('x = ', x0)
46
47     print ('\nerro encontrado')
48     print ('erro = ', erro)

```

Fonte: Exemplo 9.1 do livro

Figura 4.12: Resultado do código fonte de Cálculo Numérico



```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

PS C:\Mariana - UENF\PLP\Python\Códigos> & c

Total de Iterações
15

Valor de x
x = -5.51214599609375

erro encontrado
erro = 6.103515625e-05
PS C:\Mariana - UENF\PLP\Python\Códigos> █
```

Fonte: Exemplo 9.1 do livro



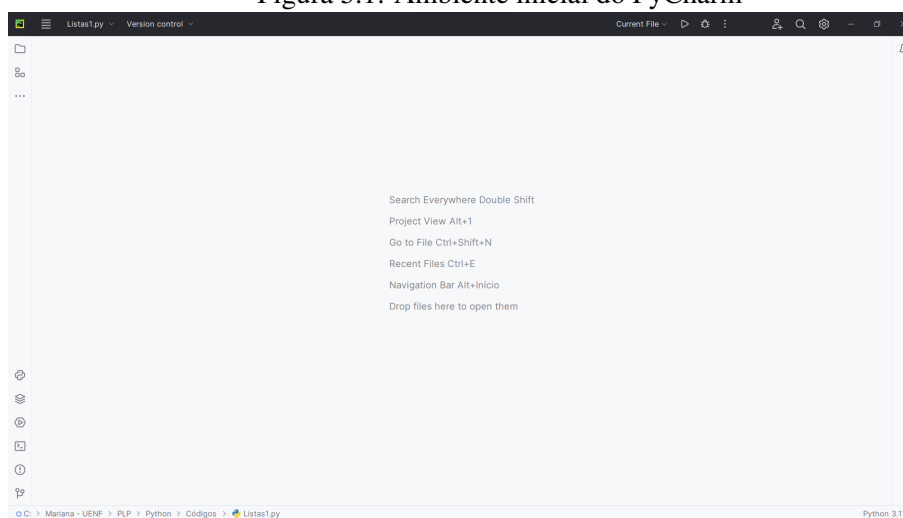
5. Ferramentas existentes e utilizadas

Neste capítulo serão apresentadas duas ferramentas que foram consultadas e utilizadas para realizar este livro, e usadas nas aplicações:

5.1 PyCharm

PyCharm é um IDE Python desenvolvido pela JetBrains. Ele foi projetado especificamente para Python e fornece muitos recursos que o tornam uma ferramenta poderosa para o desenvolvimento do Python. Atualmente, se encontra na versão 2023.1.

Figura 5.1: Ambiente inicial do PyCharm



Fonte: Criado por Mariana Cossetti Dalfior

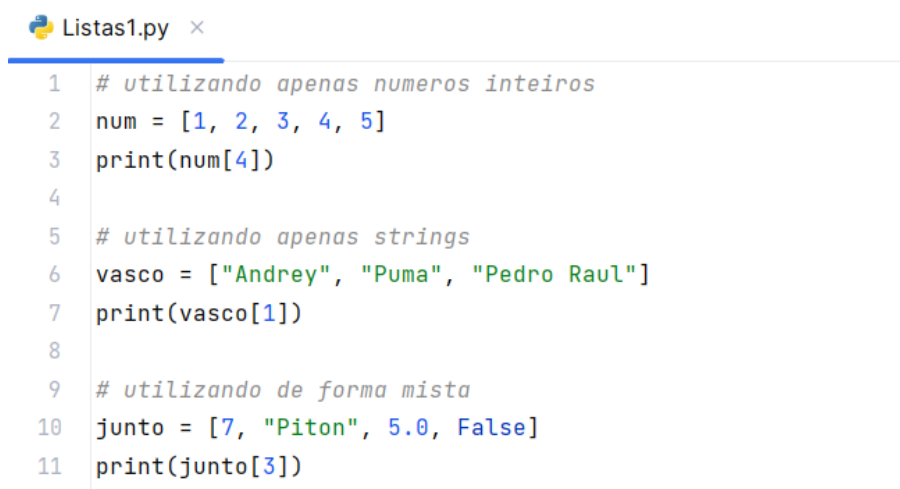
Uma das principais características do PyCharm é seu sistema de depuração. Ele permite que

you execute seu código passo a passo e observe o estado de seu programa em cada etapa. Isso é muito útil para encontrar e corrigir bugs em seu código. O PyCharm também possui um sistema de gerenciamento de projetos para organizar facilmente seu código. Ele permite que você estruture seu projeto em diferentes módulos e pacotes, tornando seu código mais fácil de gerenciar e manter.

Além disso, o PyCharm oferece suporte a vários frameworks e bibliotecas Python, como Django, Flask e NumPy. Isso significa que você pode usar o PyCharm para desenvolver uma ampla variedade de aplicativos Python, de sites a aplicativos de ciência de dados. Para baixá-lo basta apenas acessar o site da ferramenta (<https://www.jetbrains.com/pt-br/pycharm/>).

A seguir é possível observar o mesmo código fonte 5.2 e o resultado 5.3 do exemplo de código em Python mostrado acima, porém no IDE PyCharm, utilizado em 3.2.1:

Figura 5.2: Código fonte do exemplo do uso listas



```
1  # utilizando apenas numeros inteiros
2  num = [1, 2, 3, 4, 5]
3  print(num[4])
4
5  # utilizando apenas strings
6  vasco = ["Andrey", "Puma", "Pedro Raul"]
7  print(vasco[1])
8
9  # utilizando de forma mista
10 junto = [7, "Piton", 5.0, False]
11 print(junto[3])
```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 5.3: Resultado do código fonte do exemplo do uso listas

```
5
Puma
False
```

```
Process finished with exit code 0
```

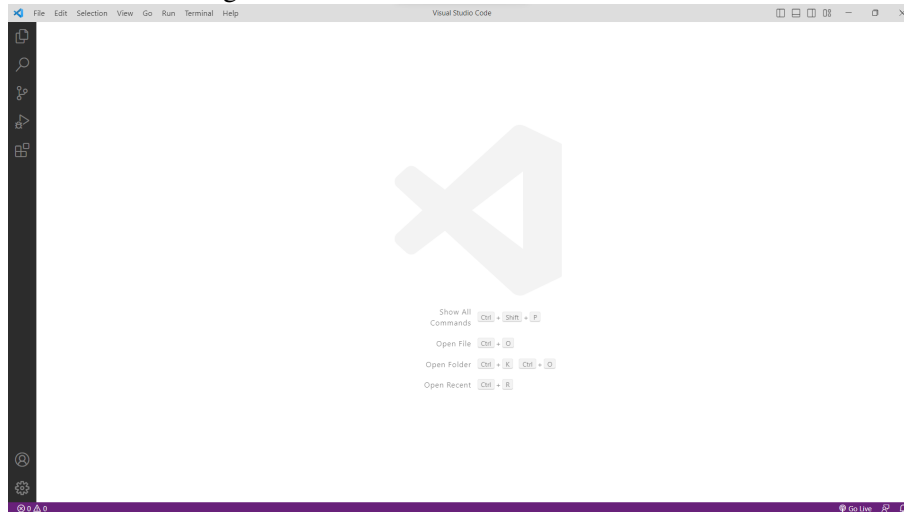
Fonte: Criado por Mariana Cossetti Dalfior

5.2 Visual Studio Code

O Visual Studio Code, comumente conhecido como VS Code, é um editor de código-fonte criado pela Microsoft que se tornou uma ferramenta essencial para muitos programadores Python. O software é leve, de código aberto e gratuito e compatível com muitas linguagens de programação, incluindo Python. Atualmente, se encontra na versão 1.79 e é a que foi utilizada para a execução

dos códigos deste livro. Além disso, pode ser utilizado em diversos sistemas operacionais como Windows, macOS e Linux.

Figura 5.4: Ambiente inicial do Visual Studio Code



Fonte: Criado por Mariana Cossetti Dalfior

O VS Code fornece um conjunto de funções que simplificam a programação em Python. Possui um sistema de realce de sintaxe que torna o código mais fácil de ler e entender. Além disso, o VS Code possui um recurso de preenchimento automático que sugere trechos de código conforme o usuário digita, economizando tempo e minimizando a chance de erros de digitação.

Um recurso notável do VS Code é sua extensibilidade. Existem milhares de extensões disponíveis que aumentam a funcionalidade do editor, como formatação automática de código, linting, depuração, teste de unidade e muito mais. Isso permite que você adapte o VS Code ao seu processo de desenvolvimento do Python. Para baixá-lo basta apenas acessar o site da ferramenta (<https://code.visualstudio.com/>).

A seguir é possível observar o código fonte 5.5 e o resultado 5.6 do exemplo de código em Python, utilizado em 3.2.1:

Figura 5.5: Código fonte do exemplo do uso listas

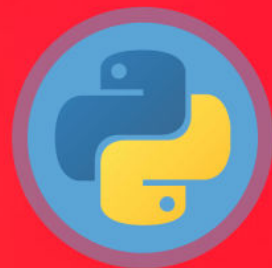
```
Listas1.py > ...
1  # utilizando apenas numeros inteiros
2  num = [1, 2, 3, 4, 5]
3  print(num[4])
4
5  # utilizando apenas strings
6  vasco = ["Andrey", "Puma", "Pedro Raul"]
7  print(vasco[1])
8
9  # utilizando de forma mista
10 junto = [7, "Piton", 5.0, False]
11 print(junto[3])
```

Fonte: Criado por Mariana Cossetti Dalfior

Figura 5.6: Resultado do código fonte do exemplo do uso listas

```
PS C:\Mariana - UENF\PLP\Python\Códigos>  
as1.py"  
5  
Puma  
False
```

Fonte: Criado por Mariana Cossetti Dalfior



6. Conclusão

Ao concluir o livro sobre Python, os leitores encontraram uma sólida base para sua jornada na programação com Python. Durante as páginas anteriores, eles exploraram a história por trás dessa linguagem de programação e como ela evoluiu ao longo do tempo, tornando-se uma das mais populares e poderosas do mundo.

O livro iniciou abordando as áreas de aplicação de Python, além de conceitos básicos, que vão desde sua sintaxe simples e legibilidade até o entendimento dos tipos de dados fundamentais, estruturas de controle de fluxo e funções. Em seguida, os conceitos avançados, explorando tópicos como programação orientada a objetos e módulos. Também abordaram assuntos mais complexos, como Classes, tipos de dados de coleção e unções geradoras.

Além disso, o livro explorou diversas aplicações de Python em diferentes campos, como operações básicas, banco de dados, programas com objetos, algoritmo de quicksort e um programa de cálculo numérico. Ao longo do livro, também foram destacadas as principais ferramentas e recursos disponíveis para programadores Python, como IDEs (Ambientes de Desenvolvimento Integrado) populares, como PyCharm e Visual Studio Code.



Referências Bibliográficas

- [Ban18] Sérgio Luiz Banin. *Python 3. Conceitos e Aplicações. Uma Abordagem Didática*. Érica, São Paulo, 2018. Citado na página 53.
- [Che18] Daniel Y. Chen. *Análise de Dados com Python e Pandas*. Novatec, São Paulo, 2018. Citado na página 12.
- [dSS19] Rogério Oliveira da Silva and Igor Rodrigues Sousa Silva. Python programming language. *TECNOLOGIAS EM PROJEÇÃO*, 2019. Citado na página 7.
- [eS20] Alan Cristoffer e Sousa. *Curso Básico de Python*. Divinópolis, MG, 2020. Citado 2 vezes nas páginas 13 e 14.
- [Fel19] Fernando Feltrin. *Python do ZERO à Programação Orientada a Objetos*. Uniorg, 2019. Citado na página 10.
- [Fel20] Fernando Feltrin. *Programação Orientada a Objetos com Python*. Uniorg, 2020. Citado na página 10.
- [Gut21] John V. Guttag. *Introduction to Computation and Programming Using Python, third edition With Application to Computational Modeling and Understanding Data*. The MIT Press, Cambridge, Massachusetts, 2021. Citado 2 vezes nas páginas 30 e 59.
- [McK18] Wes McKinney. *Python Para Análise de Dados*. Novatec, São Paulo, 2018. Citado na página 10.
- [McK23] Wes McKinney. *Python for Data Analysis Data Wrangling with Pandas, NumPy, and Jupyter*. O'Reilly Media, São Paulo, SP, 2023. Citado na página 12.
- [Mon12] Arturo Fernández Montoro. *PYTHON 3 al descubierto 2ª Edición*. RC Libros, Madrid, 2012. Citado na página 7.
- [MRA19] Bradley N. Miller, David L. Ranum, and Julie Anderson. *Python Programming in Context*. Jones e Bartlett Learning, Burlington, 2019. Citado na página 17.

-
- [Per16] Ljubomir Perkovic. *Introdução à Computação Usando Python. Um Foco no Desenvolvimento de Aplicações*. LTC, Rio de Janeiro, 2016. Citado na página 37.
- [Ram22] Luciano Ramalho. *Fluent Python : Clear, Concise, and Effective Programming*. O'Reilly Media, Sebastopol, 2022. Citado 2 vezes nas páginas 37 e 58.
- [SKSP21] Vijaya Kumara Sarma, Vimal Kumar, Swati Sharma, and Shashwat Pathak. *Python Programming*. Taylor & Francis Group, 2021. Citado na página 51.
- [Tul16] Tulchak. History of python. 2016. Citado na página 7.
- [vR10] Guido van Rossum. The python language reference. 2010. Citado na página 7.
- [Yoo22] Harry Yoon. *Python Mini Reference*. Independently Published, San Diego, Califórnia, 2022. Citado na página 15.