

Aprenda Python Básico **Rápido e Fácil de entender**



Felipe Galvão

Aprenda Python Básico - Rápido e Fácil de entender

Felipe Galvão

Prefácio

Este livro pode ser baixado gratuitamente em:

<http://felipegalvao.com.br/livros>

Você não deve alterá-lo, vendê-lo ou qualquer coisa do tipo. Mas pode imprimir para deixar na cabeceira, passar para os amigos ou dar de presente.

Quem sou eu?

Meu nome é Felipe Galvão. Sou formado em Engenharia de Produção mas sempre me dei bem com os computadores. Assim, acabei me enfiando no mundo mágico da programação. Programação é uma coisa mágica, pois com um computador, você cria um monte de coisas legais do nada.

Depois que aprendi programação, eu criei um e-commerce de camisas, que durou por alguns anos e foi uma fonte incrível de aprendizado. Depois continuei meus estudos. Estudei mais desenvolvimento web, análise de dados, trabalhei um pouco como freelancer e criei outros projetos pessoais. Vocês podem ver mais algumas coisas sobre mim em meu site pessoal:

<http://felipegalvao.com.br/>

Escrevi este livro basicamente porque escrever me ajuda a fixar o conteúdo. Python já tem bastante material bom para estudo, mas achei que seria um exercício bacana, além de poder ajudar alguém. Talvez minha didática funcione melhor com certas pessoas. De qualquer forma, boa sorte com o livro.

Sumário

1. [Introdução](#)
 1. [Objetivo](#)
 2. [Por que Python?](#)
 3. [Versão do Python](#)
 4. [Instalação do Python](#)
 5. [IDEs e Editores de Texto](#)
 6. [Digitar ou copiar o código?](#)
 7. [Conclusão](#)
2. [Print, Comentários e Tipos de Dados](#)
 1. [Funções](#)
 2. [A função print\(\)](#)
 3. [Comentários](#)
 4. [Tipos de Dados](#)
 5. [Conclusão](#)
3. [Sinais e Operações](#)
 1. [Sinais Matemáticos Básicos](#)
 2. [Sinais de Comparação](#)
4. [Conclusão](#)
5. [Variáveis](#)
 1. [Definindo variáveis](#)
 2. [A função input\(\)](#)
 3. [Conclusão](#)
6. [Strings](#)
 1. [Definindo Strings](#)
 2. [Escapando strings](#)
 3. [Strings como elas são](#)
 4. [Inserindo variáveis em uma string](#)
 5. [Concatenação](#)
 6. [Strings como listas](#)
 7. [Funções úteis para Strings](#)
 1. [capitalize\(\), lower\(\) e upper\(\)](#)
 2. [center\(\), ljust\(\) e rjust\(\)](#)
 3. [find\(\)](#)
 4. [isalnum\(\), isalpha\(\) e isnumeric\(\)](#)
 5. [len\(\)](#)

6. [replace\(\)](#)
 7. [strip\(\), rstrip\(\) e lstrip\(\)](#)
8. [Conclusão](#)
7. [Listas](#)
 1. [Listas](#)
 2. [Criando, modificando e acessando itens em uma lista](#)
 3. [Removendo Itens de uma lista](#)
 4. [Funções úteis para trabalhar com listas](#)
 1. [len\(\)](#)
 2. [max\(\) e min\(\)](#)
 3. [copy\(\)](#)
 4. [count\(\)](#)
 5. [sort\(\)](#)
 6. [reverse\(\)](#)
 7. [in](#)
 8. [split\(\) e join\(\)](#)
 5. [Conclusão](#)
8. [Tuplas](#)
 1. [Introdução](#)
 2. [Criando uma tupla e acessando seus itens](#)
 3. [Funcionalidades](#)
 4. [Conclusão](#)
9. [Dicionários](#)
 1. [Criando um dicionário, acessando e modificando suas informações](#)
 2. [Funções úteis para Dicionários](#)
 1. [get\(\)](#)
 2. [items\(\), keys\(\) e values\(\)](#)
 3. [update\(\)](#)
 3. [Conclusão](#)
10. [Sets](#)
 1. [Criando e acessando dados em Sets](#)
 2. [Algumas funções para trabalhar com sets](#)
 1. [difference\(\) e intersection\(\)](#)
 2. [copy\(\)](#)
 3. [Conclusão](#)
11. [Estruturas de Controle](#)
 1. [If, else e elif](#)
 2. [Loops - For](#)

3. [Loops - while](#)
 4. [Conclusão](#)
12. [Funções](#)
 1. [Funções Predefinidas do Python](#)
 2. [Definindo uma função](#)
 3. [args e kwargs](#)
 4. [Conclusão](#)
13. [Módulos](#)
 1. [Criando e importando seu primeiro módulo](#)
 2. [Um pouco mais sobre criação e importação](#)
 3. [Módulos do Python](#)
 4. [Conclusão](#)
14. [Programação Orientada a Objetos](#)
 1. [Definindo uma classe](#)
 2. [Alterando e acessando propriedades](#)
 3. [Propriedades de Classe e de Objeto](#)
 4. [Herança](#)
 5. [Conclusão](#)
15. [Data e Tempo](#)
 1. [Data](#)
 2. [Tempo](#)
 3. [Data e Tempo](#)
 4. [Conclusão](#)
16. [Exceções](#)
 1. [O try](#)
 2. [else e finally](#)
 3. [Chamando Exceções](#)
 4. [Conclusão](#)
17. [Arquivos](#)
 1. [A função open\(\)](#)
 2. [Escrevendo em um arquivo](#)
 3. [Adicionando a um arquivo](#)
 4. [Abrindo no modo r+](#)
 5. [Conclusão](#)
18. [Palavras Finais](#)
 1. [E agora?](#)

Introdução

Objetivo

Este livro falará de programação com Python. A meta é que seja um guia simples de ser lido e seguido, com exemplo e linguagem mais claros possível, sem enrolação. Nada de história do Python, filosofia da linguagem ou qualquer coisa do tipo.

Este livro assume um conhecimento básico de utilização do computador de uma forma geral e do Terminal (ou Prompt) do seu sistema operacional.

Por que Python?

Python é uma linguagem extremamente popular, com muitas funções. Pode ser usado para desenvolvimento para web, análise de dados, programação científica, desenvolvimento para desktop e muito mais.

Muito dessa popularidade se deve também à facilidade em ler um código Python. Muitas de suas expressões e funções se assemelham bastante com a língua inglesa, e isso facilita muito na hora de aprender e entender.

Depois de aprender Python, você pode partir para outras linguagens e frameworks, ampliar seu leque de habilidades e estudar aquilo que estiver mais em linha com seus objetivos.

Versão do Python

Caso ainda não saiba, atualmente, Python tem duas versões mais utilizadas, Python 2 e Python 3. Python 3 é a versão mais atual, e o código feito em Python 2 não é 100% compatível com ela. As versões tem diferenças consideráveis.

Dito isto, Python 3 é o futuro, e não existe qualquer motivo para que você, que está começando, comece a programar com Python 2. Por isso, vamos utilizar Python 3.

Instalação do Python

Existem várias formas de instalar e utilizar Python. Vamos utilizar a distribuição Anaconda. Esta é uma distribuição do Python que já conta com vários pacotes úteis, muito utilizados em programação científica, análise de dados *etc.*

A distribuição também conta com uma IDE gratuita, o Spyder, que é uma boa opção para começarmos a programar. IDE significa Integrated Development Environment, ou Ambiente de Desenvolvimento Integrado. É, basicamente, uma ferramenta utilizada para desenvolvimento de software, com uma série de funcionalidades úteis, como completar código, destaque de elementos das linguagens, entre outros.

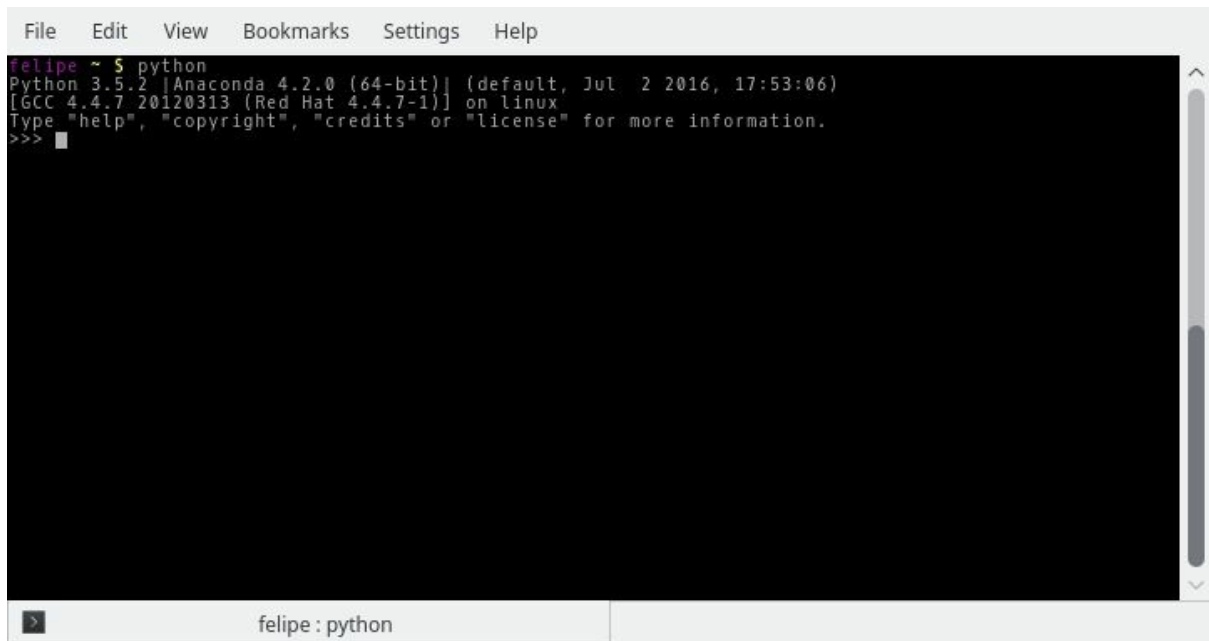
Anaconda é fácil de instalar em qualquer sistema operacional que você esteja utilizando. Para baixá-lo, acesse o link: <https://www.continuum.io/downloads>

Neste link, você encontrará opções para download para Windows, Linux e OSX. Para cada sistema operacional, existem as opções com Python 2 e Python 3. Baixe a opção com Python 3 e siga as instruções que constam na página de download.

Para verificar a instalação, abra o Terminal de seu sistema operacional (para Windows, um Prompt do Anaconda, chamado “Anaconda Prompt”, é instalado), e digite Python. Se tudo correu bem, você verá algo parecido com a tela abaixo:



```
felipe ~ $ python
Python 3.5.2 |Anaconda| (default, Jul 10 2016) [GCC 4.4.7 20120
Type "help", "copyright", "credits()", or "license()" for more
>>>
```

A screenshot of a Python terminal window. The window has a menu bar with 'File', 'Edit', 'View', 'Bookmarks', 'Settings', and 'Help'. The terminal content shows the command 'python' being executed, followed by the Python version and environment information: 'Python 3.5.2 |Anaconda 4.2.0 (64-bit)| (default, Jul 2 2016, 17:53:06) [GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux'. It then displays the prompt '>>>' with a cursor. The window title bar at the bottom shows 'felipe : python'.

Este é o prompt do Python. Nele, você pode digitar comandos com a linguagem e ver a saída destes comandos. Faça um teste e digite o código abaixo, e depois dê ENTER:

```
>>> print ( "Oi, Python") Oi , Python
```

Você verá a saída, aquilo que escreveu dentro do parênteses. Falaremos mais sobre isso nos próximos capítulos, mas você já pode ver que seu Python está funcionando.

Para sair do Terminal do Python, digite `exit()` e tecle ENTER.

IDEs e Editores de Texto

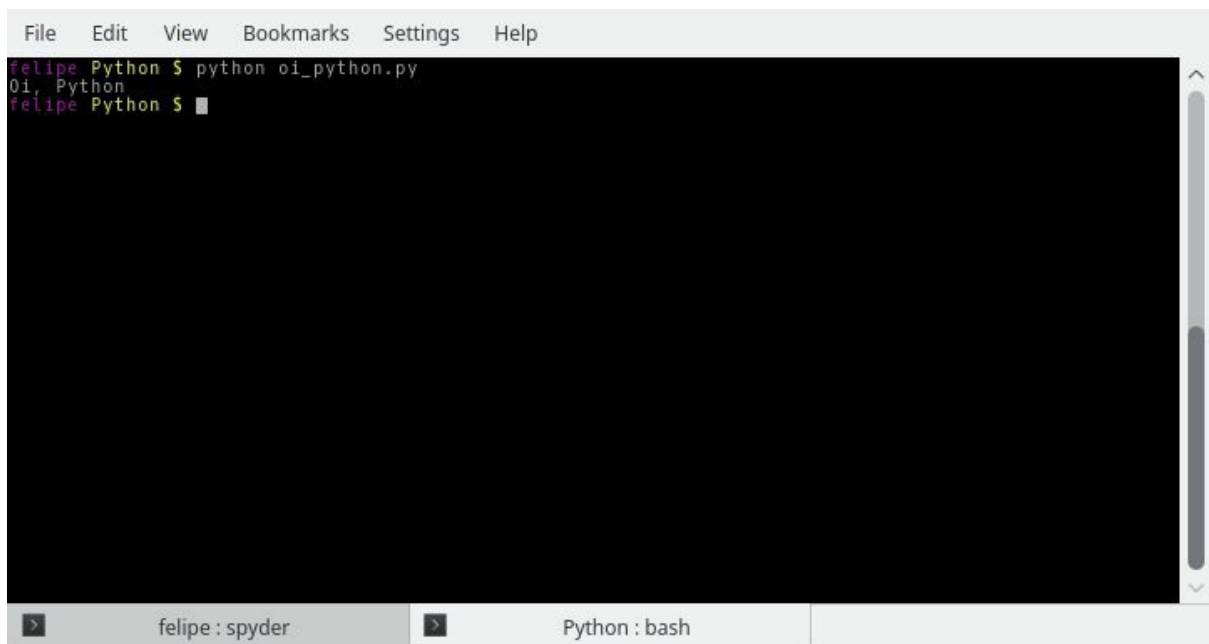
Spyder é a IDE que vem junto com a distribuição Anaconda. Ela possui muitas funções voltadas para análise de dados, mas também pode ser utilizada para criação de qualquer outro arquivo de código em Python.

Você também pode utilizar qualquer editor de texto ou, se tiver, outra IDE de sua preferência.

E com isto, podemos falar sobre a outra forma de executar código Python, que é através de arquivos .py. Abra o seu IDE ou editor preferido, crie um novo arquivo e escreva nele o código que utilizamos no Terminal, logo acima. Apenas para relembrar:

```
print ( "Oi, Python")
```

Salve este arquivo com o nome oi_python.py em uma pasta de sua preferência. Abra o Terminal, vá até a pasta onde salvou o arquivo e execute o comando “python oi_python.py”. A saída será a mesma que quando executamos o comando no Terminal do Python, como mostra a imagem abaixo:



```
felipe Python $ python oi_python.py
Oi, Python
felipe Python $
```

E é desta forma que é feita a programação de projetos e softwares mais complexos.

Ao longo do livro, tentarei sempre manter o formato de digitar o código em um arquivo .py para rodar. O nome do arquivo fica a seu critério, qualquer coisa com a extensão .py funciona. O que você tem que inserir em seu arquivo ficará com a fonte em **negrito**, e a saída que você deverá ver ao executar o código ficará com a fonte em *itálico*, e será precedido por um sinal “>”.

Faremos desta forma, primeiro pois fica fácil eu colocar todo o código em um arquivo .py, separado por capítulo, para compartilhar com vocês. E também, é mais simples para alguns dos tópicos mais complexos, como blocos de código em estrutura de controle, definição de funções, entre outras coisinhas que necessitam de mais linhas e indentação do código para funcionar corretamente. Abaixo, só um exemplo de um possível código, para que você entenda melhor como ficará a demonstração do mesmo ao longo dos capítulos:

```
nome = input("Digite aqui o seu nome: ") print("Olá, %s" % nome)
> Digite aqui o seu nome: Felipe
> Olá, Felipe
```



Digitar ou copiar o código?

Isso normalmente é uma preferência pessoal, e eu não vou ditar nenhuma regra sobre como você deve aprender. Você pode pegar o código que está no livro, copiar e colar, ou pode digitá-lo para depois rodar.

Para muitas pessoas (para mim, inclusive), digitar o código ajuda a gravar. Outras preferem copiar para passar mais rápido pelo livro e eventualmente voltar em conceitos que sejam mais importantes ou difíceis de gravar para reforçar. Faça da forma que você achar melhor.

Todo o código do livro encontra-se no seguinte repositório do Github:

https://github.com/felipegalvao/codigo_livro_aprenda_python_basico

Conclusão

Com isto, encerra-se o capítulo introdutório. Aqui, basicamente, introduzimos um pouco o assunto e preparamos o terreno para a programação em si, que ocorrerá a partir do capítulo 2.

Print, Comentários e Tipos de Dados

Vamos agora escrever nossos primeiros programas e aprender um pouco sobre sinais e operações no Python.

Funções

Primeiramente, apenas para não prejudicar o entendimento do conteúdo que virá a frente, vamos falar um pouquinho, de forma bem superficial, sobre funções. Isto pois usaremos algumas funções já definidas pelo Python para trabalhar com os tipos de dados que veremos.

Funções são blocos, pedaços de códigos predefinidos que podem ser reaproveitados. Funções podem levar parâmetros ou não. Parâmetros são informações passadas para a função, que utilizam este parâmetro para realizar algum cálculo ou transformação.

Funções podem retornar um valor final para ser utilizado posteriormente ou apenas exibir informações.

A forma usual de chamar uma função é através de seu nome, com os parâmetros dentro de parênteses. Caso a função não receba parâmetros, deve-se incluir os parênteses, mas sem nada dentro deles. A função `print`, por exemplo, recebe como parâmetro o que será exibido por ela.

A função `print()`

Já vimos um pouco da função “`print()`” no capítulo passado de Introdução. A função “`print()`”, basicamente, “imprime” na tela, ou seja, exibe, aquilo que está dentro dos parênteses.

Atente que o funcionamento desta função mudou do Python 2 para o Python 3. No Python 3, “`print()`” é, exclusivamente, uma função. Veremos mais sobre funções em um capítulo posterior, mas isso quer dizer que a única forma de chamar o “`print()`” é como vimos acima, através do seu nome seguido de parênteses, com os parâmetros da função dentro dos parênteses, quando necessário.

No Python 2, você também poderia chamar o “`print()`” como uma declaração, sem os parênteses. Isso eventualmente causava certa confusão e ambiguidade, e por isso, foi mudado no Python 3, visando aumentar a consistência da linguagem e acabar com as dúvidas sobre este assunto. Mas fique esperto, pois em muitas referências você poderá ver o código da seguinte forma:

```
print "Oi, Python"
```

No Python 3, isso não funciona mais, e retornará um erro. Por isso, fique esperto. Função `print`, só com parênteses, conforme o exemplo abaixo:

```
print("Oi, Python")
```

Comentários

Outra coisa importante que você também vai ver muito em códigos por aí são os comentários. Comentários são auto-explicativos. São anotações no código que não são rodadas junto com o próprio, e servem para ilustrar, explicar ou incluir informações adicionais nele. No Python existem duas formas de comentários. O primeiro, o comentário de uma linha, deve ser antecedido por “#”, e transforma em comentário tudo que está a partir dele, até o final daquela linha. E ele não precisa estar no começo da linha, ele pode começar logo após alguma linha de código. Veja exemplos:

```
# Isto é um comentário e não será executado pelo Python print("Oi Python")  
print("Explicando comentários") # isto também é um comentário e não será executado
```

Coloque este código em um novo arquivo e salve com o nome que quiser, e ao rodar, você verá que apenas os prints serão exibidos no seu Terminal.

Tipos de Dados

O Python permite trabalhar com diferentes tipos de dados. Um destes tipos é o tipo **String**. Vamos passar por cada tipo, explicando cada um deles:

- Strings - são dados definidos entre aspas duplas (") ou simples ('), representando texto, que normalmente é exibido por um programa ou exportado por ele.
 - Ex: "Oi Python"
- Números Inteiros (Integer) - são números sem casas decimais, números inteiros. São definidos simplesmente como um número inteiro comum.
 - Ex: 5, 31, 150
- Números decimais (float) - números com casas decimais, são definidos com o número e as casas decimais após o ponto.
 - Ex: 5.67, 99.99
- Boolean - utilizado para representar os conceitos de Verdadeiro ou Falso. Booleans só podem assumir estes dois valores, representados por "True" e "False" (mas sem as aspas)
 - Ex: True, False

Temos também outros tipos de dados, como Listas (lists), Dicionários (dictionaries) e Tuplas (tuples), entre outros. Estes são tipos especiais, que veremos em capítulos posteriores, mas servem basicamente para armazenar outros tipos de dados.

Conclusão

E assim, encerramos este capítulo. Nele, vimos um pouquinho da sintaxe básica, como criar comentários em seu código e também os tipos de dados que o Python permite que você trabalhe.

Sinais e Operações

Sinais Matemáticos Básicos

Vamos agora falar um pouco de sinais e operações em Python. Para fazer a representação das operações, vamos utilizar números inteiros (integer), mas todas elas também funcionam com números decimais (float). Para as operações matemáticas básicas, o Python possui os seguintes sinais:

Sinal	Operação	Exemplo	Definição
+	Soma	<pre>>>> 4+5 9</pre>	Soma dois números
-	Subtração	<pre>>>> 6-3 3</pre>	Subtrai dois números
*	Multiplicação	<pre>>>> 7*8 56</pre>	Multiplica dois números
/	Divisão	<pre>>>> 12/4 3.0</pre>	Divide dois números
%	Resto da Divisão	<pre>>>> 63 % 10 3</pre>	Retorna o resto da divisão inteira entre dois números
**	Potenciação	<pre>>>> 5**3 125</pre>	Eleve o número antes do sinal à potência do número após

Abaixo, veremos os exemplos em código, para que você possa confirmar o funcionamento de cada operador:

```
print(4 + 5) print(6 - 3) print(7 * 8) print(12 / 4) print(63 % 10) print(5**3)
> 9
> 3
> 56
> 3.0
> 3
> 125
```

Sinais de Comparação

Existem também os sinais de comparação. Estes sinais retornarão True ou False, indicando se uma determinada comparação é Verdadeira ou Falsa. True e False são valores especiais em Python, um tipo de dado chamado Boolean, onde os únicos valores possíveis são estes dois. Vamos aos sinais de comparação:

Sinal	Operação	Exemplo	Definição
==	Igual a	>>> 4 == 5 False >>> 4 == 4 True	Indica se dois valores são iguais
!=	Diferente de	>>> 6 != 3 True >>> 5 != 5 False	Indica se dois valores não são iguais
>	Maior que	>>> 9 > 8 True >>> 3 > 5 False	Indica se o número anterior ao sinal é maior que o posterior
>=	Maior ou igual a	>>> 10 >= 10 True >>> 13 >= 15 False	Indica se o número anterior ao sinal é maior ou igual ao posterior
<	Menor	>>> 63 < 65 True >>> 150 < 140 False	Indica se o número anterior ao sinal é menor que o posterior
<=	Menor ou igual a	>>> 5 <= 5 True >>> 12 <= 8 False	Indica se o número anterior ao sinal é menor ou igual ao posterior

E novamente, abaixo, exemplos em código e a saída de cada uma destas comparações:

```
print(4 == 5) print(4 == 4)  
print(6 != 3) print(6 != 6)  
print(9 > 8) print(3 > 5)  
print(10 >= 10) print(13 >= 15)  
print(63 < 65) print(150 < 140)  
print(5 <= 5) print(12 <= 8)  
> False > True  
> True > False  
> True > False  
> True > False  
> True > False  
> True > False
```

Conclusão

Neste capítulo vimos os operadores utilizados pelo Python para as operações matemáticas básicas e para a comparação entre dois valores. Nada muito diferente do que você está acostumado na matemática usual do dia a dia, com algumas exceções.

Variáveis

Neste capítulo, falaremos de variáveis. As variáveis são elementos que nos permitem guardar valores para reutilizar posteriormente. Imagine que você fez algum cálculo complexo com dados recuperados de algum outro lugar. Agora imagine que você deseja usá-lo posteriormente em seu programa. Não faz sentido fazer recuperar todas as informações e fazer o cálculo todo novamente. Para isso, usamos as variáveis.

Definindo variáveis

Para definir uma variável no Python, basta digitar um nome para a própria, o sinal de igual e o valor que se deseja que ela armazene.

Uma variável em Python pode armazenar praticamente qualquer coisa. Números, strings, listas, dicionários.

Os nomes de variáveis em Python precisam começar com uma letra ou com um underscore “_”. Números podem ser usados se não estiverem no começo.

Variáveis em Python também não possuem tipos definidos. Desta forma, uma mesma variável que armazena inicialmente uma string, pode depois armazenar um número inteiro, ou uma lista. Vamos então a um exemplo de definição de variáveis:

```
a = 3
b = 7
print(a+b)
a = "Agora uma string"
print(a)
> 10
> Agora uma string
```

Você também pode designar a uma variável o valor de outra variável. Prosseguindo com o exemplo acima:

```
b = a
print(b)
> Agora uma string
```

Também é possível definir mais de uma variável de uma vez. Caso o valor seja igual para todas, utilize o sinal de igual entre os nomes das variáveis. Caso seja diferente, use vírgulas entre os nomes das variáveis e entre os valores de cada uma delas. Veja os exemplos abaixo:

```
# Definição de múltiplas variáveis com valores iguais x = y = z = 10
```

```
print(x) print(y) print(z)
# Definição de múltiplas variáveis com valores diferentes x, y, z = 10, 20, 30
print(x) print(y) print(z)
> 10
> 10
> 10
> 10
> 20
> 30
```

Lembre-se que vimos no capítulo anterior o sinal de comparação “==”. É muito importante não confundir o sinal de comparação “==” e o sinal de atribuição “=”. Para definir uma variável, use sempre “=”, e para fazer uma comparação, saber se dois valores ou variáveis são iguais, use sempre “==”.

A função input()

Aqui, aproveitamos para apresentar a função “input()”. Esta função é utilizada para captar informações enviadas pelo usuário. Ela pode receber um parâmetro, que será uma mensagem exibida ao usuário antes que ele insira o que deseja.

A entrada inserida pelo usuário pode ser salva e utilizada posteriormente. No exemplo abaixo, a entrada digitada pelo usuário é salva na variável “nome” e depois é impressa, juntamente com a mensagem desejada. No exemplo, logo após a exibição da mensagem “Olá, qual o seu nome?”, o programa aguardará o input do usuário. Após digitar seu nome, você deve apertar o Enter, para que o código prossiga.

```
nome = input("Olá, qual o seu nome?\n") print("Olá, %s" % nome)
> Olá, qual o seu nome?
Felipe
> Olá, Felipe
```

Conclusão

Neste capítulo vimos como utilizar variáveis. Variáveis são essenciais na programação, pois você pode guardar valores inputados pelo usuário ou armazenar o resultado de algum cálculo feito, eliminando a necessidade de refazer este cálculo sempre que quiser utilizar seu resultado.

Basicamente, uma variável é definida escolhendo seu nome, colocando um sinal de “=”, e após destes, o valor que a variável irá assumir.

Strings

Já vimos um pouco sobre strings no capítulo 2, sobre tipos de dados que o Python utiliza. Resgatando a definição que usamos, são dados definidos entre aspas duplas (“) ou simples (‘), representando texto, que normalmente é exibido por um programa ou exportado por ele.

Resumindo e simplificando, basicamente, string é um texto, podendo conter letras, números, símbolos.

Definindo Strings

Apenas para deixar o capítulo completo, vamos relembrar como definimos uma string. Uma string é definida como tudo que se encontra entre aspas simples (') ou duplas ("). Qual você usa é uma questão de gosto e opinião pessoal. Desta forma vamos aos primeiros exemplos:

```
string1 = "Oi Python"
string2 = 'Tchau Python'
print(string1) print(string2)
> Oi Python > Tchau Python
```

Desta forma, vimos acima as duas principais formas de definir uma string. Existe uma terceira forma, que é a definição de strings entre três aspas triplas. Neste caso, caracteres escapados (veremos o que são em alguns instantes) assumem seu significado correto, e quebras de linha, seja no terminal ou na linha de código também são interpretadas como tal no output. Veja o exemplo abaixo, onde cada quebra de linha foi feita dando ENTER após cada sentença. A string só se encerra ao fechá-la com o mesmo conjunto de aspas simples ou duplas utilizado para iniciá-la:

```
string_grande = '''Olá. Esta é uma string grande no Python.
Aqui você pode usar " ou '
Caracteres são escapados como se espera.
É a terceira forma de definir uma string, apesar de não ser tão usual....
\t testando o TAB para finalizar'''

> Olá. Esta é uma string grande no Python.
> Aqui você pode usar " ou '
> Caracteres são escapados como se espera.
> É a terceira forma de definir uma string, apesar de não ser tão usual....
> testando o TAB para finalizar
```

Escapando strings

Ok, tá bom. Mas e se eu quiser criar uma string com aspas duplas e simples no meio dela? Bem, se você usa aspas duplas em sua string e quiser usar aspas simples nela, ou vice-versa, não há qualquer problema. Basta colocá-los junto com a string e ela continuará sendo válida. Veja:

```
string3 = "A cantora Sinnead O'Connor"  
string4 = 'Alfredo disse "Corram aqui para ver isso!"'  
print(string3) print(string4)  
> A cantora Sinnead O'Connor > Alfredo disse "Corram aqui para ver isso!"
```

Agora, caso você queira utilizar as duas em uma string, ou caso sempre defina suas strings com aspas duplas e queira usar aspas duplas em uma string específica, temos que escapar este caracter. Escapar? Sim, escapar. Escapar significa inserir em uma string caracteres que tenham algum significado especial, como uma nova linha, um caracter de tabulação (TAB) ou as aspas simples ou duplas.

Para escapar um caracter em Python utilizamos a contrabarra (\) antes do elemento que desejamos escapar. Vejamos os exemplos:

```
string5 = "Alfredo disse \"Corram aqui para ver isso!\""  
string6 = 'Sinnéad O'Connor disse "Nothing compares 2 u"'  
print(string5) print(string6)  
> Alfredo disse "Corram aqui para ver isso!"  
> Sinnéad O'Connor disse "Nothing compares 2 u"
```

E se eu quiser usar uma contrabarra na minha string? Fácil, basta escapar ela também. Ao definir uma string, duas contrabarras (\\) viram uma contrabarra na saída:

```
string7 = "Estou escapando uma \\  
print(string7)  
> Estou escapando uma \
```

Retirado da documentação e traduzido por mim mesmo, segue abaixo uma tabelinha com sequências escapadas e seus significados:

--	--

Sequência Escapada	Significado
\\	Contrabarra
\'	Aspas simples
\"	Aspas duplas
\a	Caracter de controle ASCII Bell (BEL)
\b	Caracter ASCII Backspace (BS)
\f	Caracter ASCII Formfeed (FF)
\n	Caracter ASCII Linefeed (LF) - este cria uma nova linha no output
\r	Caracter ASCII Carriage Return (CR)
\t	Caracter ASCII Tab Horizontal (TAB)
\v	Caracter ASCII Tab Vertical (VT)
\ooo	Caracter ASCII com valor octal "ooo"
\xhh	Caracter ASCII com valor hex de "hh"

Strings como elas são

Uma outra forma de representar strings é antecedendo sua definição com a letra “r”. Isto gera o que chamamos de raw string literals, onde a string é exibida exatamente como ela foi definida, e não substituindo caracteres escapados por seus significados no output. Vejamos o exemplo abaixo para entender:

```
string8 = r"Utilizamos \n para inserir uma nova linha na string"
print(string8)
> Utilizamos \n para inserir uma nova linha na string
```

Desta forma, podemos escrever strings sem se preocupar com caracteres escapados ou qualquer coisa do tipo.

Inserindo variáveis em uma string

Outra situação bastante recorrente: tenho um determinado valor armazenado em uma variável e quero exibi-lo juntamente com outros caracteres de texto. Posso fazer isso de diferentes formas. A primeira delas é utilizando o sinal de “%”, que é um caractere de formatação de strings no Python.

Apesar de permitir utilizações mais complexas para formatação de strings, sua utilização mais básica é através do %s para incluir outras strings em uma string, e %d para incluir um número inteiro (Integer). Vejamos exemplos dos dois:

```
nome = "Felipe"
idade = 30
print("Meu nome é %s " % nome) print("Tenho %d anos" % idade)
> Meu nome é Felipe > Tenho 30 anos
```

Também podemos incluir números decimais (float) em strings, usando %f. Vejamos algumas possibilidades no exemplo abaixo. Primeiro, um exemplo sem qualquer formatação:

```
a = 30.46257
print("Formatando decimais: %f" % a)
> Formatando decimais: 30.462570
```

Se colocamos um ponto e a quantidade de casas decimais, definimos a quantidade de algarismos a serem impressos após o ponto:

```
print("Formatando decimais: %.2f" % a) print("Formatando decimais: %.3f" % a)
> Formatando decimais: 30.46
> Formatando decimais: 30.463
```

Você pode inserir mais de um valor na string, colocando-os em ordem dentro de parênteses, separados por vírgulas. Esta estrutura se chama Tupla, e em um capítulo posterior, a veremos em detalhes:

```
nome = 'Felipe'
idade = 30
print("Meu nome é %s e tenho %d anos" % (nome, idade))
```


Meu nome é Felipe e tenho 30 anos

Concatenação

Outra forma de inserir valores em strings é através da concatenação. O sinal de concatenação no Python é o “+”. Através dele, juntamos strings com variáveis e outras strings. Vejamos no exemplo abaixo:

```
nome = 'Felipe'
print("Olá, meu nome é " + nome)
> Olá, meu nome é Felipe
```

Esta forma tem um porém. Vejamos agora o que acontece se incluirmos a variável idade, um número inteiro, nesta string, por concatenação:

```
nome = 'Felipe'
idade = 30
print("Olá, meu nome é " + nome + " e tenho " + idade + " anos.")
> Traceback (most recent call last): > File "<stdin>", line 1, in <module> > TypeError: Can't
convert 'int' object to str implicitly
```

A função print não faz a conversão de um número inteiro para uma string automaticamente. Desta forma, para incluir esta variável, precisamos converter o número inteiro para uma string usando a função str, e passando dentro do parênteses o valor a ser convertido. Veja:

```
print("Olá, meu nome é " + nome + " e tenho " + str(idade) + " anos.")
> Olá, meu nome é Felipe e tenho 30 anos.
```

E no caso de números decimais? Bem, se você não quer formatá-lo, basta chamar a função str da mesma forma que no exemplo anterior:

```
valor = 503.78987
print("O valor é " + str(valor))
> O valor é 503.78987
```

Mas, se quiser formatá-la antes, teremos que usar outra função, a função format. Seus parâmetros são o valor a ser formatado e o formato das casas decimais, tal qual vimos na formatação de decimais nos exemplos anteriores, onde passamos o ponto e o número de casas decimais após o mesmo. Vejamos no exemplo:

```
print("O valor é " + format(valor, '.2f'))
> O valor é 503.79
```

Note que a função `format` já retorna uma string, não havendo aqui a necessidade de converter o resultado em string usando a função `str`.

Strings como listas

Em Python, toda string é tratada como uma lista. Ainda falaremos mais de listas em capítulos posteriores, mas resumindo, uma lista é uma estrutura semelhante aos arrays, para quem vem de outras linguagens de programação. São um conjunto de dados armazenados em uma única estrutura. E em Python, uma string é uma lista, ou uma sequência de caracteres.

Uma das situações que isso desencadeia é que podemos acessar qualquer caractere da string através de seu índice, ou index, que é a sua posição na string. A primeira letra tem índice 0 (zero), e a última é igual a quantidade de caracteres da string menos um. Vejamos:

```
string9 = "Olá, meu nome é Felipe"
print(string9[0]) print(string9[3]) print(string9[21])
> O
> ,
> e
```

No exemplo acima, a string que usamos tem 22 caracteres. E o que acontece se usarmos um índice que não existe? O Python retorna um erro. Vejamos:

```
print(string9[25])
> Traceback (most recent call last): > File "<stdin>", line 1, in <module> > IndexError: string
index out of range
```

No capítulo onde falaremos sobre listas veremos mais sobre como trabalhar com elas de forma mais eficiente.

Funções úteis para Strings

Aqui, vamos apresentar algumas funções úteis para strings, explicando um pouco o funcionamento de cada uma delas. Não vou entrar em detalhes de todas as funções existentes no Python, pois são muitas e a documentação da linguagem é até melhor que eu para explicar as mesmas. Vou apenas apresentar algumas das que considero importantes e mais utilizadas e explicá-las com bons exemplos.

Chamamos as funções abaixo, como veremos nos exemplos, como métodos da própria string.

capitalize(), lower() e upper()

A função “*capitalize()*” transforma a primeira letra de uma string em maiúscula. “*lower()*” e “*upper()*” transformam, respectivamente, todas as letras em minúsculas e maiúsculas. Veja alguns exemplos:

```
string10 = "olá, meu NOME é Felipe"
print(string10.capitalize()) print(string10.upper())
> Olá, meu nome é felipe > OLÁ, MEU NOME É FELIPE
```

center(), ljust() e rjust()

A função “*center()*”, como o nome provavelmente já dá a entender, centraliza uma string para um determinado número de caracteres, utilizando um caractere a ser definido para preencher a string dos dois lados. “*ljust()*” e “*rjust()*” fazem o mesmo, mas preenchem caracteres apenas à esquerda ou à direita. Veja um exemplo com o *center*. Os outros dois são análogos:

```
string11 = "olá, meu nome é Felipe"
print(string11.center(50,"*"))
> *****olá, meu nome é Felipe*****
```

find()

Este método indica se uma determinada substring faz parte de uma string. Caso faça, ela retorna a posição onde começa esta substring. Caso não, a função retorna -1. Veja no exemplo:

```
string12 = "Olá, meu nome é Felipe"
substring1 = "meu"
print(string12.find(substring1)) substring2 = "José"
print(string12.find(substring2))
> 5
> -1
```

Você também pode definir os índices a partir dos quais a busca inicia e termina, através dos parâmetros beg (primeiro após a string a ser encontrada) e end (segundo após a string, logo após o parâmetro beg), conforme o exemplo abaixo:

```
print(string12.find(substring1, 7)) print(string12.find(substring1, 2))
> -1
> 5
```

isalnum(), isalpha() e isnumeric()

Estas funções indicam se uma determinada string é, respectivamente, totalmente formada por caracteres alfanuméricos, alfabéticos (só contem letras) e números. Se as strings tiverem pelo menos um caracter que invalide cada condição, estas funções retornarão “False”. Vale notar que espaço em branco na string torna uma string inválida para qualquer uma destas funções. Vamos com muitos exemplos, para podermos entender bem. Primeiro, uma string só com letras:

```
string13 = "Felipe"
print(string13.isalnum()) print(string13.isalpha()) print(string13.isnumeric())
> True
> True > False
```

Agora, uma string só com números:

```
string14 = "1234"
print(string14.isalnum()) print(string14.isalpha()) print(string14.isnumeric())
```

```
> True  
> False > True
```

E uma com os letras e números:

```
string15 = "Felipe1234"  
print(string15.isalnum()) print(string15.isalpha()) print(string15.isnumeric())  
> True  
> False > False
```

len()

A função “*len()*” retorna a quantidade de caracteres presentes em uma determinada string. Repare que a função “ *len()* ”, diferentemente da maioria das funções que vimos, não é chamada através de uma string, mas a string é passada como um argumento para ela. Veja:

```
string16 = "Meu nome é Felipe"  
print(len(string16))  
> 17
```

replace()

A função “ *replace()* ”, como o nome já indica, substitui uma parte de uma string por outra parte, definidas nos dois primeiros argumentos da função. Veja o exemplo:

```
string17 = "Olá, meu nome é Felipe"  
print(string17.replace("Felipe", "José"))  
> Olá, meu nome é José
```

strip(), rstrip() e lstrip()

A função “ *strip()* ” remove espaços no começo e no fim das strings. Já as funções “ *rstrip()* ” e “ *lstrip()* ” removem, respectivamente, espaços à direita e à esquerda da string (r e l de right e left). Vejamos alguns exemplos:

```
string18 = " Olá, meu nome é Felipe "  
print(string18) print(string18.strip()) print(string18.rstrip()) print(string18.lstrip())
```

```
> Olá, meu nome é Felipe > Olá, meu nome é Felipe > Olá, meu nome é Felipe > Olá, meu nome  
é Felipe
```

PS: Não dá para conferir muito bem o resultado da função “rstrip()”, mas eu juro que ela funciona.

Conclusão

Existem muitas outras funções úteis relacionadas a strings no Python, mas checamos algumas das mais importantes e mais utilizadas. Vimos também como definir strings, incluir o valor de variáveis nelas e formatar números a serem inseridos. Assim como o tratamento que o Python dá às strings, tratando-as como listas.

Listas

O Python possui algumas diferentes estruturas de dados utilizados para armazenar diferentes tipos de dados. Cada um deles tem suas particularidades. Neste capítulo, veremos um pouco sobre as listas.

Listas

Se você está vindo de outra linguagem de programação, é possível que já tenha trabalhado com arrays. As listas do Python são os arrays das outras linguagens.

Se não veio, vou explicar. As listas são estruturas de dados sequenciais do Python que servem para armazenar outros diferentes tipos de dados. As listas podem armazenar qualquer tipo de dado, como números inteiros, strings, números decimais e até outras listas.

Criando, modificando e acessando itens em uma lista

Para criar uma lista, basta definir seus itens entre colchetes “[]” e separados por vírgula. Também é possível criar uma lista vazia, definindo apenas os colchetes, sem nenhum item dentro deles. Vejamos alguns exemplos:

```
alunos = ["José", "João", "Luiz"]
notas = [8.5, 9.2, 6.7]
print(alunos)
print(notas)
lista_vazia = []
print(lista_vazia)
> ['José', 'João', 'Luiz']
> [8.5, 9.2, 6.7]
> []
```

Note que os itens não precisam ser todos do mesmo tipo. Você pode misturar itens de diferentes tipos dentro de uma mesma lista.

```
lista_misturada = [12, 15.56, "Sorveteria", ["Baunilha", "Chocolate"]]
print(lista_misturada)
> [12, 15.56, 'Sorveteria', ['Baunilha', 'Chocolate']]
```

A lista acima, como você viu, possui um inteiro, um decimal, uma string e uma outra lista.

Para acessar os itens de uma lista, você utiliza o índice de cada uma delas, ou a posição que o item desejado ocupa na lista. O primeiro item é o de índice 0 (zero). Vejamos:

```
print(alunos[0]) print(notas[2])
> José
> 6.7
```

Você também pode usar índices negativos, e assim a contagem será feita do final da lista:

```
print(alunos[-1]) print(alunos[-3])
> Luiz
> José
```

Da mesma forma, também podemos alterar o valor que está nestes índices.

```
print(notas)
notas[2] = 7.7
print(notas)

> [8.5, 9.2, 6.7]
> [8.5, 9.2, 7.7]
```

E quando queremos adicionar novos itens a uma lista? Bem, nesse caso temos três funções. A primeira delas é a função “ *append()* ”. A função “ *append()* ”, chamada através de uma lista, recebe como parâmetro o item a ser adicionado ao final da lista:

```
print(alunos)
alunos.append('Alfredo') print(alunos)

> ['José', 'João', 'Luiz']
> ['José', 'João', 'Luiz', 'Alfredo']
```

E se não quisermos que o novo item fique no final? Bem, aí usamos a função “ *insert()* ”. Nesta função, indicamos o índice que o novo item deverá ficar. Os outros itens são ajustados de acordo:

```
print(alunos)
alunos.insert(1, "Daniela") print(alunos)

> ['José', 'João', 'Luiz', 'Alfredo']
> ['José', 'Daniela', 'João', 'Luiz', 'Alfredo']
```

Repare como ‘Daniela’ se tornou o item no índice 1, conforme indicamos na função “ *insert()* ”.

Por fim, temos a função “ *extend()* ”. Esta função adiciona uma lista de itens ao final de outra lista:

```
print(alunos)
novos_alunos = ['Carlos', 'Maria', 'Ana']
alunos.extend(novos_alunos) print(alunos)
```

```
> ['José', 'Daniela', 'João', 'Luiz', 'Alfredo']  
> ['José', 'Daniela', 'João', 'Luiz', 'Alfredo', 'Carlos', 'Maria', 'Ana']
```

Também é possível concatenar duas listas, utilizando apenas o sinal de “+”:

```
alunos1 = ['José', 'Daniel', 'João']  
alunos2 = ['Carlos', 'Augusto', 'Denis']  
print(alunos1 + alunos2)  
> ['José', 'Daniel', 'João', 'Maria', 'Ana', 'Carolina']
```

E por fim, também é possível usar o sinal de multiplicação para repetir itens de uma lista:

```
print(notas*2)  
> [8.5, 9.2, 7.7, 8.5, 9.2, 7.7]
```

Removendo Itens de uma lista

Bem, e para remover? Para remover itens de uma lista, temos as funções “*remove()*” e *pop()*. A função “*remove()*”, apaga o item com base em seu valor:

```
print(alunos)
alunos.remove('João') print(alunos)

> ['José', 'Daniela', 'João', 'Luiz', 'Alfredo', 'Carlos', 'Maria', 'Ana']
> ['José', 'Daniela', 'Luiz', 'Alfredo', 'Carlos', 'Maria', 'Ana']
```

A função “*pop()*”, remove um item da lista com base no seu índice. Caso nenhum argumento seja passado, ele remove o último item. A função também retorna o item removido:

```
print(alunos)
aluno_removido = alunos.pop() print(aluno_removido) print(alunos)
aluno_removido = alunos.pop(2) print(aluno_removido) print(alunos)

> ['José', 'Daniela', 'Luiz', 'Alfredo', 'Carlos', 'Maria', 'Ana']
> Ana
> ['José', 'Daniela', 'Luiz', 'Alfredo', 'Carlos', 'Maria']
> Luiz
> ['José', 'Daniela', 'Alfredo', 'Carlos', 'Maria']
```

E se temos dois itens com o mesmo valor e usamos a função “*remove()*”? Neste caso, a função remove apaga apenas a primeira ocorrência deste valor:

```
alunos = ['José', 'Denis', 'Daniela', 'Carla', 'Carlos', 'Augusto', 'Denis']
print(alunos)
alunos.remove('Denis') print(alunos)

> ['José', 'Denis', 'Daniela', 'Carla', 'Carlos', 'Augusto', 'Denis']
> ['José', 'Daniela', 'Carla', 'Carlos', 'Augusto', 'Denis']
```

Repare no exemplo acima que a lista possuía o nome Denis duas vezes, mas apenas a primeira ocorrência, que era o segundo item da lista, foi removido.

As listas também têm uma funcionalidade interessante, que é a de extrair partes da lista, o que em inglês é chamado de “slicing”. Para fazer isso, usamos uma notação semelhante à de acessar itens dentro das listas, mas informamos o intervalo que desejamos acessar. Vejamos o exemplo para que fique mais claro:

```
print(alunos)
print(alunos[0:2]) print(alunos[2:4]) print(alunos[2:5])
> ['José', 'Daniela', 'Carla', 'Carlos', 'Augusto', 'Denis']
> ['José', 'Daniela']
> ['Carla', 'Carlos']
> ['Carla', 'Carlos', 'Augusto']
```

Com esta notação, o Python pega os itens no intervalo definido dentro dos colchetes, iniciando o intervalo no primeiro número e terminando no segundo, mas sem incluir o segundo.

Se omitimos um dos números, ele parte do princípio da lista ou vai até o final dela.

```
print(alunos[:3]) print(alunos[3:])
> ['José', 'Daniela', 'Carla']
> ['Carlos', 'Augusto', 'Denis ']
```

Também é possível utilizar números negativos. Desta forma, a contagem é feita a partir do final:

```
print(alunos[1:-1]) print(alunos[2:-2])
['Daniela', 'Carla', 'Carlos', 'Augusto']
['Carla', 'Carlos']
```


Funções úteis para trabalhar com listas

Entre as funções interessantes que podemos utilizar com as listas estão:

len()

A função “ *len()* ”, assim como ocorre quando utilizada com strings, retorna a quantidade de itens em uma lista:

```
print(len(alunos))  
> 6
```

max() e min()

As funções “ *max()* ” e “ *min()* ”, como os nomes já devem indicar, retornam o valor máximo e o mínimo em uma lista:

```
print(max(notas)) print(min(notas))  
> 9.2  
> 7.7
```

copy()

A função “ *copy()* ” é a forma correta de criar uma cópia de uma lista. Definir uma nova lista igual a uma lista prévia (utilizando o sinal de “=”), apesar de parecer intuitivo, não funciona, pois isto cria apenas uma referência à lista. Vejamos primeiramente a forma errada:

```
alunos = ["José", "João", "Luiz", "Carlos", "Afonso"]  
alunos_backup = alunos print(alunos_backup) alunos.clear()  
print(alunos_backup)  
> ['José', 'Daniela', 'Carla', 'Carlos', 'Augusto', 'Denis']  
> []
```

Repare como a lista “*alunos_backup*” possuía os itens iguais à lista *alunos*, mas ficou vazia após usarmos a função “ *clear()* ” para remover todos os

itens da lista alunos. De nada adiantou fazer o backup desta forma.

Vejam agora a forma correta de criar uma cópia, usando a função “`copy()`” (vamos aproveitar e recriar a lista “alunos”, que ficou vazia após o último exemplo):

```
alunos = ['José', 'Daniela', 'Carla', 'Carlos', 'Augusto', 'Denis']
alunos_backup = alunos.copy() print(alunos_backup) alunos.clear() print(alunos_backup)
> ['José', 'Daniela', 'Carla', 'Carlos', 'Augusto', 'Denis']
> ['José', 'Daniela', 'Carla', 'Carlos', 'Augusto', 'Denis']
```

Agora, mesmo limpando a primeira lista, a lista de backup permanece inalterada. Vamos apenas recriar a lista “alunos” novamente agora, para o caso de a usarmos em exemplos próximos.

```
alunos = ['José', 'Daniela', 'Carla', 'Carlos', 'Augusto', 'Denis']
```

count()

A função “`count()`” retorna a quantidade de vezes que um determinado item ocorre em uma lista. Basta passar este item que se deseja contar como parâmetro da função. Vamos adicionar alguns nomes repetidos na lista “alunos” com a função “`extend()`” e checar o funcionamento da função “`count()`”:

```
alunos.extend(['Daniela', 'Felipe', 'Carla', 'Daniela']) print(alunos.count('Daniela'))
> 3
```

sort()

A função “`sort()`” ordena uma lista. Mas ela tem uma particularidade, e você deve tomar cuidado. Ela não retorna nenhum valor, mas sim ordena a lista permanentemente. Caso você precise manter os dados na ordem que estavam e deseje apenas fazer uma exibição da lista ordenada, deve-se fazer uma cópia da lista:

```
alunos.sort()
```

```
print(alunos)
```

```
> ['Augusto', 'Carla', 'Carla', 'Carlos', 'Daniela', 'Daniela', 'Daniela', 'Denis', 'Felipe', 'José']
```

reverse()

A função “*reverse()*” reverte a ordem dos elementos de uma lista. Assim como a função “*sort()*”, ela executa a operação permanentemente na lista onde é chamada, e por isso, deve-se tomar cuidado ao utilizá-la:

```
alunos.reverse() print(alunos)
```

```
> ['José', 'Denis', 'Daniela', 'Daniela', 'Daniela', 'Carlos', 'Carla', 'Carla', 'Augusto']
```

in

Por último, é possível verificar se um determinado item consta ou não na lista. Para isso, usamos o “in”, da seguinte forma:

```
print("José" in alunos) print("Felipe" in alunos)
```

```
> True
```

```
> False
```

split() e join()

Estas são funções com propósitos opostos, e elas na verdade operam em “strings”. Mas elas são funções que se relacionam com listas (ou outros tipos de sequências), e preferi aguardar o capítulo de listas para falar sobre elas.

A função “*split()*” retorna uma lista a partir de uma string e de um separador definido, que pode ser qualquer coisa, mas por padrão é um espaço.

Já a função “*join()*” retorna uma string a partir de uma sequência, com os itens da sequência divididos por um determinado separador.

```
alunos_string = "; ".join(alunos) print(alunos_string) alunos_lista = alunos_string.split("; ")  
print(alunos_lista) > José; Denis; Daniela; Daniela; Daniela; Carlos; Carla; Carla; Augusto >  
['José', 'Denis', 'Daniela', 'Daniela', 'Daniela', 'Carlos', 'Carla', 'Carla', 'Augusto']
```

Conclusão

E assim concluímos o capítulo sobre listas. Vimos que as listas são estruturas para armazenar dados de diferentes tipos. Vimos também como criar, acessar dados e modificar uma lista. E por fim, vimos algumas funções bastante úteis para serem usados com as listas. No próximo capítulo veremos uma estrutura semelhante, a tupla.

Tuplas

Após vermos as listas no capítulo anterior, neste, veremos as tuplas, ou tuples, no inglês.

Introdução

As tuplas assemelham-se bastante às listas. São estruturas que armazenam dados de diferentes tipos, em forma de lista.

A grande diferença é que ao contrário de uma lista, uma tupla é imutável. Você não pode redefinir elementos de uma tupla, alterar seus valores ou qualquer coisa do tipo.

Criando uma tupla e acessando seus itens

A forma para se criar uma tupla é bastante semelhante à de uma lista também. Só que ao invés de usarmos colchetes, usamos parênteses. E da mesma forma, separamos seus elementos com vírgulas:

```
tupla1 = ("Gato", "Cachorro", "Papagaio", "Tartaruga") print(tupla1)
> ('Gato', 'Cachorro', 'Papagaio', 'Tartaruga')
```

Para acessar seus elementos, também fazemos da mesma forma que com as listas, ou seja, através de colchetes indicando a posição do elemento desejado:

```
print(tupla1[2])
> Papagaio
```

Além disso, também é possível pegar apenas uma parte da tupla, usando a notação com colchetes e os números de início e fim desejados divididos por “:”, assim como já vimos com as listas. Esta funcionalidade é denominada “slicing”:

```
print(tupla1[1:3])
> ('Cachorro', 'Papagaio')
```

Também é permitido criar uma tupla vazia, definindo apenas os parênteses, sem qualquer elemento dentro:

```
tupla_vazia = () print(tupla_vazia)
> ()
```

Bem, como dissemos que uma tupla é imutável, vamos provar. Vamos tentar alterar o valor de um item de uma tupla, assim como fizemos no capítulo de listas:

```
tupla1[1] = "Elefante"

> Traceback (most recent call last): > File "<stdin>", line 1, in <module> > TypeError: 'tuple'
object does not support item assignment
```


Repare como o próprio Python reporta o erro. Uma tupla não suporta a designação do valor de um item.

Igualmente, não é possível remover itens de uma tupla. Duas coisas que você pode fazer são criar uma nova tupla sem o item desejado e apagar uma tupla completamente, através da função “*del()*”:

```
del(tupla1)
```

```
print(tupla1)
```

```
> Traceback (most recent call last): > File "<stdin>", line 1, in <module> > NameError: name  
'tupla1' is not defined
```

Funcionalidades

Algumas das funções que vimos no capítulo anterior também funcionam com as tuplas, como “ *len()* ”, “ *max()* ” e “ *min()* ”, por exemplo:

```
tupla2 = (8.3, 9.4, 3.3, 7.5, 7.6) print(max(tupla2)) print(min(tupla2)) print(len(tupla2))
> 9.4
> 3.3
> 5
```

Também é possível transformar uma tupla em uma lista e vice-versa, através das funções *list* e *tuple*:

```
lista1 = list(tupla1) print(lista1) lista2 = ["José", "Afonso", "Carlos", "Luiz"]
tupla3 = tuple(lista2) print(tupla3)
> ["Gato", "Cachorro", "Papagaio", "Tartaruga"]
> ('José', 'Afonso', 'Carlos', 'Luiz')
```

Conclusão

E assim, concluímos o capítulo sobre tuplas, que nada mais são do que listas imutáveis. Vimos que muito do que fazemos com as listas também funcionam com as tuplas e as formas de criá-las e acessar as informações contidas nelas. No próximo capítulo falaremos sobre dicionários, estruturas muito interessantes para armazenar dados de uma forma diferente das listas e tuplas.

Dicionários

Os dicionários do Python, ou dictionaries (também referenciados como dicts), são mais uma estrutura para armazenar dados. Entretanto, o que difere significativamente os dicionários das listas é que no dicionário, o índice é um nome que você define, ao invés dos índices numerados que as listas possuem.

Criando um dicionário, acessando e modificando suas informações

Para criar um dicionário, usamos as chaves ({}). Os elementos seguem o formato “chave” : “valor”. Vejamos no exemplo para que fique mais claro:

```
aluno = {"nome": "José", "idade": 20, "nota": 9.2}
print(aluno)
> {'nome': 'José', 'idade': 20, 'nota': 9.2}
```

Como já se deve imaginar, é possível criar um dicionário vazio, definindo apenas as chaves, sem elementos dentro:

```
dict_vazio = {}
print(dict_vazio)
> {}
```

Para acessar os dados de um dicionário, usamos o mesmo formato das listas. Porém, como o índice é nomeado, usaremos o nome para resgatar o valor, ao invés do índice numérico:

```
print(aluno["nome"]) print(aluno["idade"])
> José > 20
```

Se tentarmos acessar uma chave que não existe no dicionário, o Python retorna um erro:

```
print(aluno["Peso"])
> Traceback (most recent call last): > File "<stdin>", line 1, in <module> > KeyError: 'Peso'
```

Para modificar um valor, novamente, fazemos como nas listas. Basta indicar a chave que se deseja alterar e definir o novo valor. A inclusão de um novo item no dicionário é feita da mesma forma, utilizando uma chave que ainda não foi utilizada no mesmo:

```
aluno["nota"] = 8.2
aluno["peso"] = 74
print(aluno)
> {'idade': 20, 'nome': 'José', 'nota': 8.2, 'peso': 74}
```

Repare como a chave “peso” e seu valor foram adicionados, além da alteração no valor da chave “nota”.

Para remover uma determinada chave e seu valor, usamos o “del”. Para apagar todas as chaves e valores, usamos o método “clear”, chamado através do dicionário:

```
del aluno["idade"]  
print(aluno) aluno.clear() print(aluno)  
> {'nome': 'José', 'nota': 8.2, 'peso': 74}  
> {}
```

Funções úteis para Dicionários

Algumas das funções que utilizamos com as listas também funcionam com os dicionários. É o caso da função “ *len()* ”, por exemplo, que retorna a quantidade de pares chave : valor do dicionário (vamos aproveitar para redefinir o dicionário “aluno” que limpamos no último exemplo:

```
aluno = {"nome": "José", "idade": 20, "nota": 9.2}
print(len(aluno))
> 3
```

Outra funcionalidade que também funciona com dicionários é a verificação da existência de uma chave através do “ *in* ”:

```
print("idade" in aluno) print("peso" in aluno)
> True
> False
```

Mas temos outras funções úteis para trabalhar com dicionários. Vamos ver algumas delas.

get()

A função “ *get()* ” é muito útil para evitar erros de chave inexistente. Ela recebe dois parâmetros, que são a chave a ser buscada e um valor de retorno. Ela retorna o valor de desta chave se ela existir no dicionário, e se não existir, retorna aquilo que definimos como valor de retorno, ou None se nenhum valor de retorno for definido:

```
aluno = {"nome": "José", "idade": 20, "nota": 9.2}
print(aluno.get("nome")) print(aluno.get("peso")) print(aluno.get("peso", "Não existe"))
> José
> None
> Não existe
```

items(), keys() e values()

A função “ *items()* ” retorna uma lista de tuplas com os pares “chave” : valor. Já a função “ *keys()* ” retorna uma lista apenas com as chaves do dicionário. E a função “ *values()* ” retorna uma lista apenas com os valores dos itens do dicionário:

```
print(aluno.items()) print(aluno.keys()) print(aluno.values())  
> dict_items([('idade', 20), ('nota', 9.2), ('nome', 'José')]) > dict_keys(['idade', 'nota', 'nome']) >  
dict_values([20, 9.2, 'José'])
```

update()

A função “ *update()* ” recebe um dicionário como parâmetro e insere os pares deste dicionário parâmetro no dicionário através do qual a função é chamada. Caso existam chaves coincidentes, o primeiro dicionário é atualizado com os valores do dicionário passado como parâmetro:

```
aluno_original = {"nome": "José", "idade": 20, "nota": 9.2}  
aluno_update = {"peso": 75, "nota": 8.7}  
aluno_original.update(aluno_update) print(aluno_original)  
> {'nome': 'José', 'idade': 20, 'nota': 8.7, 'peso': 75}
```

Como podem ver, a chave “peso” foi adicionada e o valor da chave “nota” foi modificado.

Conclusão

Neste capítulo aprendemos sobre os dicionários e sua principal diferença para listas. Vimos como criar um dicionário, atualizar e acessar suas informações e também funções que podem vir a ser muito úteis ao trabalhar com eles. No próximo capítulo, veremos rapidamente os sets.

Sets

Os sets são a última estrutura para armazenar dados que veremos. Elas não são tão comumente utilizadas, mas têm uma função.

Os sets são listas ou coleções não ordenadas de itens únicos.

Criando e acessando dados em Sets

Para criar um set, você pode utilizar a função “`set()`”, que transforma sequências (uma lista, por exemplo) em sets, ou você pode definir os valores do set separados entre vírgulas (mas sem formatar em pares “chave”: valor, ou você criará um dicionário):

```
lista1 = ["Luiz", "Alfredo", "Felipe", "Alfredo", "Joana", "Carolina", "Carolina"]
set1 = set(lista1) print(set1)
set2 = {"cachorro", "gato", "papagaio", "cachorro", "papagaio", "macaco", "galinha"}
print(set2)
```

```
> {'Felipe', 'Alfredo', 'Joana', 'Carolina', 'Luiz'}
> {'cachorro', 'papagaio', 'gato', 'galinha', 'macaco'}
```

Como vimos anteriormente, strings em Python também são tratadas como listas. Desta forma, também podemos usar a função “`set()`” para criar um set a partir de uma string:

```
set3 = set("papagaio") print(set3)
> {'g', 'p', 'i', 'a', 'o'}
```

Sets são muito usados para remover valores duplicados de listas. A forma usualmente utilizada para isso é a do exemplo abaixo:

```
lista2 = ["Luiz", "Alfredo", "Felipe", "Alfredo", "Joana", "Carolina", "Carolina"]
lista_sem_duplicatas = list(set(lista2)) print(lista_sem_duplicatas)
> ['Felipe', 'Joana', 'Luiz', 'Alfredo', 'Carolina']
```

Para remover valores de um set, use a função “`remove()`”:

```
print(set2)
set2.remove("cachorro") print(set2)

> {'gato', 'galinha', 'papagaio', 'macaco', 'cachorro'}
> {'gato', 'galinha', 'papagaio', 'macaco'}
```

Algumas funções para trabalhar com sets

Dentre as funções que também funcionam com sets, estão o “*len()*”, que retorna a quantidade de itens no mesmo, e o uso do “*in*” para verificar a existência de um item no set:

```
print(len(set2)) print("gato" in set2) print("elefante" in set2)
> 4
> True
> False
```

Além destas, o Python oferece algumas outras funções interessantes para o trabalho com sets. Vejamos alguns exemplos.

difference() e **intersection()**

As funções “*difference()*” e “*intersection()*” retornam, como o nome já diz, a diferença e a interseção entre dois sets, ou seja, itens que estão em um set e não estão no outro ou itens que estão nos dois sets:

```
set4 = {"Luiz", "Alfredo", "Joana", "Felipe", "Mauro"}
set5 = {"Joana", "Carolina", "Afonso", "Carlos", "Mauro"}
print(set4.difference(set5)) print(set4.intersection(set5))
> {'Luiz', 'Felipe', 'Alfredo'}
> {'Joana', 'Mauro'}
```

copy()

Da mesma forma que vimos para listas, para criar uma cópia de um set, deve-se utilizar a função “*copy()*”. Criar um novo set apenas utilizando o sinal de igual faz apenas uma referência. Primeiro, vejamos como uma simples definição com o sinal de igual não resolve:

```
set_backup = set4
print(set_backup) set4.clear() print(set_backup)
> {'Mauro', 'Alfredo', 'Felipe', 'Joana', 'Luiz'}
> set()
```

Agora, a forma que funciona:

```
set4 = {"Luiz", "Alfredo", "Joana", "Felipe", "Mauro"}  
set_backup = set4.copy() print(set_backup) set4.clear() print(set_backup)  
> {'Alfredo', 'Mauro', 'Luiz', 'Joana', 'Felipe'}  
> {'Alfredo', 'Mauro', 'Luiz', 'Joana', 'Felipe'}
```

Conclusão

E assim, terminamos o capítulo sobre sets, e também acabamos com as estruturas utilizadas pelo Python para armazenar dados. Vimos como criar sets e utilizá-los para remover duplicatas de listas. Também vimos como manipulá-los e algumas funções interessantes para se utilizar ao trabalhar com sets. No próximo capítulo falaremos sobre Estruturas de Controle.

Estruturas de Controle

Estruturas de Controle são utilizadas para organizar e adicionar complexidade à um programa. Elas servem diferentes finalidades, e vamos ver cada uma delas em detalhes.

If, else e elif

O “*if*” é uma estrutura de controle que executa um determinado bloco de código baseado em uma condição. Se a condição estipulada for verdadeira, o bloco logo abaixo será executado. Este bloco de código a ser executado deve estar indentado, ou seja, com espaços em seu início, e sempre com a mesma quantidade de espaços (veja o exemplo abaixo para entender), para que o Python possa identificar onde ele começa e termina. Após a exibição da mensagem, se você digitar Felipe, uma mensagem será exibida, pois a condição estipulada na definição do bloco “*if*” é verdadeira. Caso digite qualquer outra coisa, nada será exibido. Abaixo, considere que rodamos o programa duas vezes. Na primeira, digitamos como nome “Felipe”, e na segunda, digitamos “Maria”:

```
nome = input("Olá, qual o seu nome?\n") if nome == "Felipe": print("Olá, %s" % nome)
> Digite aqui seu nome: Felipe > Olá, Felipe
> Digite aqui seu nome: Maria
```

Bem, e se eu quiser imprimir outra mensagem caso a condição não seja verdadeira? Bem, aí usamos o “*else*”. O *else* é sempre executado se a condição estipulada no *if* não for verdadeira:

```
nome = input("Olá, qual o seu nome?\n") if nome == "Felipe": print("Olá, %s" % nome)
else: print("Olá, visitante")
> Digite aqui seu nome: Felipe > Olá, Felipe
> Digite aqui seu nome: Maria > Olá, visitante
```

Por fim, e se quisermos testar mais de uma condição? Aí temos o “*elif*”. Podemos usar o “*elif*” quantas vezes quisermos, para testar quantas condições forem necessárias. Se nenhuma delas for verdadeira, o bloco “*else*”, localizado ao final, é executado:

```
if nome == "Felipe": print("Olá, Felipe") elif nome == "João": print("Oi, João") elif nome
== "Carlos": print("E aí, Carlos?") else: print("Olá, visitante")
> Digite aqui seu nome: Felipe > Olá, Felipe
> Digite aqui seu nome: Maria > Oi, Maria
> Digite aqui seu nome: Carlos > E aí, Carlos?

> Digite aqui seu nome: Marcia > Olá, visitante
```


Loops - For

Os loops são estruturas de controle para repetição. Imagine se eu quisesse escrever um programa que imprima todos os números de 1 a 10, ou que imprima separadamente cada item de uma determinada lista. Você poderia criar uma linha com um print para cada número, ou para cada item. Mas isso seria repetitivo, tedioso. Tem de haver uma maneira melhor de fazer isso. E têm. Para facilitar, o Python possui o loop “*for*”. Este loop itera sobre os elementos de uma sequência, executando o código no bloco indentado logo abaixo. Pode ser uma string, uma lista. Vamos executar o código do exemplo:

```
lista = ["Alfredo", "Mario", "José", "Carolina", "Joana", "Luiza"]
for nome in lista: print(nome)
> Alfredo > Mario > José > Carolina > Joana > Luiza
```

Você também pode iterar sobre dicionários, mas existem diferentes formas. Como sabemos, dicionários possuem pares chave-valor. Desta forma, é possível iterar sobre apenas um ou sobre os dois. Vamos ver cada uma das formas abaixo:

```
aluno = {"nome": "Maria", "idade": 20, "nota": 9.2}

print("Exemplo 1a - iterando sobre chaves") for chave in aluno: print(chave)
print("\nExemplo 1b - iterando sobre chaves") for chave in aluno.keys(): print(chave)
print("\nExemplo 2 - iterando sobre valores") for valor in aluno.values(): print(valor)
print("\nExemplo 3 - iterando sobre ambos") for chave, valor in aluno.items(): print(chave
+ " - " + str(valor))
> Exemplo 1a - iterando sobre chaves > nota > idade > nome
> Exemplo 1b - iterando sobre chaves > nota > idade > nome
> Exemplo 2 - iterando sobre valores > 9.2
> 20
> Maria
> Exemplo 3 - iterando sobre ambos > nota - 9.2
> idade - 20
> nome - Maria
```

No caso da impressão dos números, podemos usar a função “*range()*”. Ela cria uma sequência de números inteiros, podendo receber um ou dois

parâmetros. Recebendo um parâmetro, ela cria o intervalo que vai de 0, até o número passado na função, excluindo o próprio número. Caso sejam passados dois parâmetros, o intervalo começa no primeiro e termina no segundo:

```
print("Range com 1 parâmetro") for i in range(5): print(i)
print("Range com 2 parâmetros") for i in range(3,10): print(i)
> Range com 1 parâmetro > 0
> 1
> 2
> 3
> 4
> Range com 2 parâmetros > 3
> 4
> 5
> 6
> 7
> 8
> 9
```

Logo, se a ideia for repetir um certo bloco de código um determinado número de vezes, basta usar o “ *range()* ” com o número de repetições desejadas. E como você pode perceber através dos 2 exemplos, o nome que vem após o “ *for* ” pode ser qualquer nome que siga as regras de nomenclatura de variáveis. Este nome é simplesmente o nome que se dará a cada item dentro da lista, que será diferente em cada iteração.

Dois comando importantes presentes nos loops são o “ *break* ” e o “ *continue* ”. Esses comandos alteram o fluxo natural de um loop. No caso do “ *break* ”, ele para a execução do loop. Já no “ *continue* ”, ele não executa mais nada naquele ciclo de iteração e passa para o próximo. Vamos dar uma olhada em um exemplo para que fique mais claro:

```
print("Exemplo break") for i in range(1,11): if i % 5 == 0: break print(i)
print("Exemplo continue") for i in range(1,11): if i % 5 == 0: continue print(i)
> Exemplo break > 1
> 2
> 3
> 4
> Exemplo continue > 1
> 2
> 3
> 4
> 6
> 7
```

> 8 > 9

No primeiro exemplo, no caso do “ *break* ”, o loop deveria imprimir todos os números de 1 a 10. Entretanto, inserimos uma condição tal que, se o valor de “ *i* ” fosse divisível por 5, ele executaria o comando “ *break* ”. Dessa forma, quando o valor de “ *i* ” chegou a 5, o comando “ *break* ” foi executado e nada mais foi impresso.

A mesma lógica foi seguida no segundo exemplo. Entretanto, ao invés do “ *break* ”, o comando “ *continue* ” foi executado, fazendo com que o loop pulasse para a próxima iteração, e não imprimindo o número 5.

Loops - while

O loop while é semelhante ao for, no sentido que ele leva a repetição de um bloco de código. Entretanto, ele executa este código enquanto uma certa condição definida for verdadeira. Vejamos abaixo um exemplo semelhante ao de cima:

```
contador = 0
while contador < 5: print(contador) contador = contador + 1

> 0
> 1
> 2
> 3
> 4
```

Este exemplo vai imprimindo a variável contador, e no final do bloco de código, aumenta seu valor em 1. Conforme a condição, enquanto o contador for menor que 5, o bloco de código dentro dele será executado. Quando o valor chega em 5, a condição não é mais verdadeira e o bloco não é mais executado.

Têm de se tomar cuidado com o loop while, pois se você definir uma condição que nunca se torne falsa, seu código fica executando sem parar. Se no exemplo acima nós não incluíssemos a última linha, que incrementa o contador em 1, o código rodaria para sempre, imprimindo zero infinitas vezes. Caso você se veja nesta situação em algum código, basta apertar Ctrl + C, e o código será interrompido.

Outra possibilidade é definir como condição para execução do loop “*while True*”, e dentro do loop, definir alguma condição onde se execute o “*break*”. Vejamos um exemplo um pouco mais complexo, misturando o “*while True*” com uma estrutura “*if / else*”:

```
while True: nome = input("Digite seu nome ou sair para terminar o programa: ") if nome
== "sair": break else: print("Olá, %s" % nome)
> Digite seu nome ou sair para terminar o programa: Felipe > Olá, Felipe > Digite seu nome ou
sair para terminar o programa: Maria > Olá, Maria > Digite seu nome ou sair para terminar o
programa: sair
```

Conclusão

E assim, passamos por todas as estruturas de controle. Vimos como elas permitem adicionar mais complexidade ao seu código, executando diferentes blocos de código baseado em determinadas condições ou repetindo blocos de código, evitando código repetido. No próximo capítulo veremos em detalhes uma parte do Python que já estamos usando desde os primeiros capítulos, que são as funções.

Funções

Neste capítulo entraremos em detalhes sobre como funcionam as funções no Python. Como já vimos no início do livro, funções são blocos de código que podem ser reutilizados quantas vezes forem necessárias, para que você não tenha que ficar escrevendo o mesmo código várias vezes. Funções podem receber ou não parâmetros, para que sejam utilizadas em seu código.

Funções Predefinidas do Python

O Python já vem com muitas funções predefinidas. Algumas delas, já estamos usando há algum tempo, como “*len()*” e “*print()*”, por exemplo. Temos também algumas funções presentes em módulos, que precisam ser importados para serem utilizados.

Módulos nada mais são do que códigos e funções já definidos para serem empacotados e reutilizados em outros projetos ou outras partes de um mesmo projeto. Existem módulos já definidos pelo próprio Python, você pode criar os seus ou utilizar módulos criados por outras pessoas. Mais detalhes sobre módulos ficam para um próximo capítulo.

Definindo uma função

A estrutura básica para definir uma função é através da declaração “def”, depois o nome da função e parênteses. Caso a função não receba parâmetros, os parênteses ficarão vazios. Se receber, os parâmetros devem ser nomeados na definição. Primeiro, vejamos um exemplo de uma função sem parâmetros. Primeiro definimos a função, e em seguida a chamamos:

```
def print_ola_tres_vezes(): print("Ola Python") print("Ola Python") print("Ola Python")
print_ola_tres_vezes()
> Ola Python
> Ola Python > Ola Python
```

Funções podem ter valores retornados. Definimos o valor a ser retornado pela função através do comando “return”. O que é retornado pode ou não estar entre parênteses. Vamos ver um exemplo de uma função com um parâmetro e um valor retornado:

```
def numero_ao_cubo(numero): valor_a_retornar = numero * numero * numero
return(valor_a_retornar)
numero = numero_ao_cubo(4) print(numero)

> 64
```

Se você tentar chamar esta função acima sem o devido argumento, o Python retornará um erro, como podemos ver abaixo:

```
numero = numero_ao_cubo()
> Traceback (most recent call last): > File "11-funcoes.py", line 15, in <module> > numero =
numero_ao_cubo() > TypeError: numero_ao_cubo() missing 1 required positional argument:
'numero'
```

Uma das coisas que você pode fazer é definir um valor padrão, que será o valor daquele argumento se nenhum valor para ele for incluído na chamada da função. Para definir um argumento padrão, definimos o valor padrão na hora de definir a função, da seguinte forma:

```
def print_ola(nome="estranho"): print("Olá, " + nome)
print_ola("Priscilla") print_ola()
```



```
> Olá, Priscilla > Olá, estranho
```

Assim, vemos que, se providenciamos o argumento para a função, ele imprime o que é passado. Se não, ele imprime o nome padrão, definido na chamada da função.

Você também pode chamar uma função e nomear os argumentos que está passando. Este é o único caso onde você pode passar os argumentos em uma ordem diferente da ordem definida na criação da função:

```
def print_infos(nome, idade): print("Olá, meu nome é %s e tenho %d anos" % (nome, idade))
print_infos(idade=30, nome="Felipe")
> Olá, meu nome é Felipe e tenho 30 anos
```

Caso você não nomeie os argumentos que está passando, eles têm de estar na mesma ordem em que foram definidos.

args e kwargs

Existe uma forma no Python de definir as funções para que recebam uma quantidade variável de argumentos, além da forma normal de definição de argumentos. Esta forma é através da definição de argumentos com um ou dois asteriscos antes de seus nomes. Por convenção, costuma-se utilizar os nomes *args* e **kwargs*, mas não existe uma restrição do Python para que sejam utilizados apenas estes nomes. Você pode usar o nome que quiser.

Ambas as formas permitem que seja passado para a função um número qualquer de argumentos. A diferença entre eles é que com um asterisco, ou, por convenção, **args*, estes argumentos não são nomeados, enquanto que se definirmos a função com o formato de dois asteriscos, utilizam-se argumentos nomeados na chamada da função. Vejamos um exemplo de cada vez:

```
def print_tudo_2_vezes(*args): for parametro in args: print(parametro + "! " + parametro + "!")
print_tudo_2_vezes("Olá", "Python", "Felipe")
> Olá! Olá!
> Python! Python!
> Felipe! Felipe!
```

Como vemos, no caso do *args*, *você pode acessar os parâmetros passados como uma lista*. No caso do **kwargs*, os argumentos vem em forma de dicionário, tendo as mesmas propriedades que vimos anteriormente. Desta forma, para iterar sobre os argumentos passados, podemos fazer como no exemplo abaixo:

```
def print_info(**kwargs): for parametro, valor in kwargs.items(): print(parametro + " - " + str(valor))
print_info(nome="Felipe", idade=30, nacionalidade="Brasil")
> nacionalidade - Brasil > nome - Felipe > idade - 30
```

Por fim, você pode misturar estas funcionalidades entre si e com argumentos nomeados. Vejamos apenas um exemplo para finalizar:

```
def print_info(nome, idade, **kwargs): print("Nome: " + nome) print("Idade: " + str(idade))
print("\nInformações adicionais:") for parametro, valor in kwargs.items(): print(parametro
```

```
+ " - " + str(valor))  
print_info(nome="Felipe", idade=30, nacionalidade="Brasil", telefone="999998888")  
> Nome: Felipe  
> Idade: 30  
  
> Informações adicionais: > telefone - 999998888  
> nacionalidade - Brasil
```

Como vimos, a função acima usa parâmetros nomeados, que seriam os padrões, esperados pela função, além da funcionalidade do ****kwargs** para receber parâmetros adicionais.

Conclusão

Neste capítulo, vimos em detalhes o que são e como trabalhar com funções. As diferentes formas de definir uma função, parâmetros a serem recebidos, como chamar uma função e também como passar um número variável de parâmetros. No próximo capítulo, falaremos de módulos.

Módulos

Módulos são uma forma de organizar o seu código. Basicamente, você define suas funções em um arquivo e pode reutilizá-lo facilmente em outros projetos, podendo inclusive disponibilizá-lo para a utilização de outras pessoas.

Você pode definir seus próprios módulos, utilizar módulos que já vem com o Python ou instalar módulos de terceiros para utilizar em seus projetos. Vamos começar pela definição de nossos próprios módulos.

Criando e importando seu primeiro módulo

Vamos criar nosso primeiro módulo. Primeiro, salve o código abaixo em seu editor de texto com o nome “modulo_exemplo.py”:

```
def modulo_print():  
    print("Oi, eu sou a primeira função dentro do módulo")  
def modulo_print_com_nome(nome): print("Olá %s, eu sou a segunda função dentro do  
módulo")
```

Agora, tente rodar o código abaixo em outro arquivo:

```
modulo_exemplo.modulo_print() modulo_print()
```

```
> Traceback (most recent call last): > File "12-modulos.py", line 8, in <module> >  
modulo_exemplo.modulo_print() > NameError: name 'modulo_exemplo' is not defined
```

Você recebe um erro. O Python não sabe onde procurar por esta função. Você a definiu em outro arquivo (“modulo_exemplo.py”) e não indicou neste acima onde esta função está. E como indicamos? Fazendo um “import”. O “import” permite que você adicione em um arquivo código que foi definido em outro arquivo. Vamos fazer na prática para que fique claro:

```
import modulo_exemplo  
modulo_exemplo.modulo_print() modulo_exemplo.modulo_print_com_nome("Felipe")  
> Oi, eu sou a primeira função dentro do módulo > Olá Felipe, eu sou a segunda função dentro  
do módulo
```

Agora sim, você pode usar as funções que definiu em seu módulo, o arquivo “modulo_exemplo.py”. Repare que ao fazer o “import” não devemos colocar o formato .py ao final do nome do módulo. Incluímos a extensão “.py” apenas na hora de salvar.

Existem duas outras formas de importar uma função para utilizar em seu código. Vamos à primeira:

```
from exemplo_modulo import modulo_print  
modulo_print()  
modulo_print_com_nome("Felipe")
```

```
> Oi, eu sou a primeira função dentro do módulo > Traceback (most recent call last): > File "12-  
modulos.py", line 15, in <module> > modulo_print_com_nome("Felipe") > NameError: name  
'modulo_print_com_nome' is not defined
```

Nesta forma, você importa apenas a função que deseja usar. Neste caso, como vemos, importamos a função “*modulo_print()*” e a chamamos sem precisar colocar o nome do módulo e o ponto antes do nome da função. Repare também que ela importa somente esta função. Se tentamos usar a outra função que está definida no módulo, temos um erro. Se quisermos usar a outra função, temos de importá-la também.

A outra forma de importar que vou mostrar não é recomendada. Vamos ver o exemplo e depois explico:

```
from modulo_exemplo import *  
  
modulo_print()  
modulo_print_com_nome("Felipe")  
> Oi, eu sou a primeira função dentro do módulo > Olá Felipe, eu sou a segunda função dentro  
do módulo
```

Esta forma é uma variação da forma anterior. Nela, o asterisco indica que tudo que está no módulo deve ser importado. Parece prático, mas esta forma nos expõe a sérios riscos. E se importarmos outro módulo, criado por outra pessoa, com uma função que tenha o mesmo nome? Em projetos grandes, que utilizem alguns módulos, esta situação tem uma chance boa de acontecer. E isto causa todo tipo de comportamento inesperado em seu programa. Por isso, esta é uma forma não recomendada de importar funções de outros módulos para seu programa.

Como tudo que é feito no terminal, você pode proceder da mesma forma em arquivo .py, usando “import nome_do_modulo” ou “from nome_do_modulo import funcao”.

Um pouco mais sobre criação e importação

E se eu quiser criar um módulo em outra pasta. Isso certamente permite uma maior organização do seu código e de seus arquivos. Bem, precisamos entender então, como o Python procura pelos módulos quando você usa o “import”. Basicamente, o Python procura da seguinte forma:

1. Primeiro, procura no diretório atual; este seria o diretório que contém o script onde você está importando o módulo ou o diretório no qual você iniciou o terminal do Python;
2. Segundo, o Python procura em cada diretório definido no PYTHONPATH, que é uma variável do sistema que possui uma série de diretórios onde módulos podem ser encontrados;
3. Por último, o Python checa o diretório padrão da instalação do Python que está sendo utilizada. No caso do Linux, por exemplo, costuma ser “usr/local/lib/python”;

Para conferir os diretórios presentes no seu PYTHONPATH, basta entrar no terminal do Python e seguir os comandos abaixo:

```
import sys  
sys.path
```

Isto irá imprimir uma lista com todos os diretórios presentes no PYTHONPATH. Um módulo definido em um destes diretórios será importado através dos métodos que vimos acima.

Módulos do Python

O Python já vem com uma série de módulos predefinidos. Para estes, não é preciso definir nada, apenas importar e usar. Um exemplo é o módulo sys, que usamos no último exemplo e contém funções e parâmetros específicos do sistema.

Uma lista completa de módulos padrões do Python pode ser encontrada no seguinte link:

<https://docs.python.org/3/library/>

Conclusão

Neste capítulo, vimos como módulos podem ser utilizados para a melhor organização do código. Eles permitem a divisão de um código em arquivos, permitindo a fácil reutilização dos mesmos em outros projetos. Vimos também sobre os módulos padrão do Python, como o sys, que possui diversas propriedades e funções relacionadas ao sistema em si.

Programação Orientada a Objetos

Programação Orientada a Objetos é uma forma de programar e organizar seu código. Não farei aqui uma explicação detalhada de todos os conceitos, mas vou mostrar um pouco de como trabalhar desta forma no Python.

Definindo uma classe

Para definir uma classe no Python utilizamos a declaração “class” e em seguida o nome da classe. Usamos a declaração “def __init__(self)” para definir o construtor da classe, uma função que será chamada sempre que uma instância desta classe for criada. E podemos então definir as funções, ou métodos desta classe. Vamos definir uma classe de um usuário, com um construtor e uma função qualquer:

```
class Usuario: def __init__(self, nome, email): self.nome = nome self.email = email
def diga_ola(self): print("Olá, meu nome é %s e meu email é %s" % (self.nome, self.email))
usuario1 = Usuario(nome="Felipe", email="contato@felipegalvao.com.br")
usuario1.diga_ola() print(usuario1.nome)
> Olá, meu nome é Felipe e meu email é contato@felipegalvao.com.br > Felipe
```

A declaração do `__init__` serve para, entre outras coisas, definir as propriedades que um elemento desta classe terá. Depois, basta definir os métodos como funções normais, com a diferença que o primeiro parâmetro, na hora da definição, é o “self”. Ao chamar um método em uma instância, não há necessidade de incluir o “self”. Tanto para acessar propriedades quanto para chamar métodos, usamos a instância, um ponto e o nome do atributo ou do método.

Alterando e acessando propriedades

Você também pode alterar uma propriedade de uma instância, através da simples redefinição da mesma. Adicionando ao final do código anterior:

```
usuario1.nome = "Felipe Galvão"  
print(usuario1.nome)  
> Felipe Galvão
```

Existem também funções específicas para manipulação de objetos de uma determinada classe. São as funções abaixo:

- `hasattr(objeto, propriedade)` - Checa se um objeto possui uma determinada propriedade;
- `getattr(objeto, propriedade)` - Retorna o valor de uma propriedade do objeto;
- `setattr(objeto, propriedade, valor)` - Define o valor de uma propriedade do objeto. Se esta propriedade não existe, ela é criada;
- `delattr(objeto, propriedade)` - Remove uma propriedade do objeto.

Continuando o código anterior:

```
print(hasattr(usuario1, "nome")) print(hasattr(usuario1, "idade")) print(getattr(usuario1,  
"email")) setattr(usuario1, "nome", "Felipe G.") setattr(usuario1, "idade", 30)  
print(getattr(usuario1, "nome")) print(getattr(usuario1, "idade")) delattr(usuario1,  
"idade") print(getattr(usuario1, "idade"))  
> True > False > contato@felipegalvao.com.br > Felipe G.  
> 30  
> Traceback (most recent call last): > File "exemplo_classes_1.py", line 27, in <module> >  
print(getattr(usuario1, "idade")) > AttributeError: 'Usuario' object has no attribute 'idade'
```

Nas últimas linhas, vemos que recebemos um erro, pois deletamos uma propriedade através da função “`delattr()`” e depois tentamos recuperar seu valor com o “`getattr()`”.

Agora, vamos comentar esta última linha que está nos causando este erro, apenas para que possamos continuar com a execução dos códigos que virão à frente.

Propriedades de Classe e de Objeto

Vamos voltar um pouco para podermos falar dos tipos de propriedades. As propriedades que definimos até agora, dentro do “ `__init__()` ”, são as chamadas propriedades de objeto. Estas são específicas de um determinado objeto daquela classe, como o nome e o email do usuário. Cada usuário terá um nome e email diferentes.

Entretanto, podemos definir também variáveis de classe. Estas variáveis são relacionadas à classe como um todo. Por exemplo, se quiséssemos um contador de usuários, definiríamos uma variável como contador e poderíamos incrementá-la a cada novo usuário criado. Vamos alterar nossa classe “Usuario” então, para implementarmos este contador. Depois, vamos criar um novo usuário e verificar se nosso contador está funcionando adequadamente. Vou colocar toda a definição da classe e a criação do novo usuário. Mantenha a criação do “usuario1” conforme já fizemos acima e depois rode o código todo:

```
class Usuario: contador = 0

def __init__(self, nome, email): self.nome = nome self.email = email Usuario.contador += 1

def diga_ola(self): print("Olá, meu nome é %s e meu email é %s" % (self.nome, self.email))
.
.
.

usuario2 = Usuario(nome="Jurema", email="jurema@jurema.com")
print(Usuario.contador)
> 2
```

Contador funcionando, vamos para o próximo conceito, o conceito de “herança”.

Herança

Herança é a propriedade da programação orientada a objetos através da qual uma determinada classe pode ser estendida a partir de outra, “herdando” os métodos e propriedades desejadas e adicionando outros, mais específicos. Você tem que fazer algumas alterações específicas no método `__init__`, para que sua instância mantenha os atributos da classe original e tenha seus novos atributos específicos. Vamos criar uma classe “Administrador”, que será estendida a partir da classe “Usuario” que já definimos. Para definir que uma classe será estendida a partir de outra, incluímos a classe original dentro de parênteses da nova, como se fosse um parâmetro. Depois, fazemos as mudanças no “`__init__()`”, chamando o “`__init__()`” da classe original e adicionando as propriedades específicas logo após:

```
class Administrador(Usuario):  
    def __init__(self, nome, email, chave_de_administrador):  
        Usuario.__init__(self, nome, email)  
        self.chave_de_administrador = chave_de_administrador  
    def poder_administrativo(self):  
        print("Eu tenho o poder! Minha chave é %s" %  
              self.chave_de_administrador)  
usuario_admin = Administrador(nome="Admin", email="admin@admin.com",  
                               chave_de_administrador="123456")  
usuario_admin.diga_ola()  
usuario_admin.poder_administrativo()  
> Olá, meu nome é Admin e meu email é admin@admin.com > Eu tenho o poder! Minha chave é  
123456
```

Veja que o novo “usuario_admin” tem acesso tanto aos métodos definidos na classe “Usuario” quanto ao método definido na sua própria classe, “Administrador”.

Você também pode sobrescrever um método que esteja presente na função original. Basta definir o método na nova classe com o mesmo nome da classe original. Vejamos uma nova classe e vamos sobrescrever o método “diga_ola()”, presente na classe original:

```
class MembroStaff(Usuario):  
    def __init__(self, nome, email):  
        Usuario.__init__(self, nome, email)  
    def diga_ola(self):  
        print("Olá, meu nome é %s e eu sou membro do Staff. Em que posso  
              ajudar?" % (self.nome))  
membro_staff_1 = MembroStaff(nome="Mariazinha", email="maria@zinha.com.br")  
membro_staff_1.diga_ola()
```

> Olá, meu nome é Mariazinha e eu sou membro do Staff. Em que posso ajudar?

Conclusão

Assim, fechamos o capítulo sobre programação orientada a objetos. Vimos como definir classes, criar novos objetos de uma classe, alterar as propriedades de um objeto e acessá-los. Vimos também como o conceito de “herança” é feito no Python e as particularidades das classes derivadas. No próximo capítulo falaremos sobre data e tempo no Python.

Data e Tempo

Neste capítulo veremos como trabalhar com datas e tempo no Python. Creio que não preciso mencionar o quanto isso é importante, visto a quantidade de sistemas que precisa mexer com data e tempo.

Para trabalhar com datas e tempo no Python, usaremos o módulo “datetime”, que já vem definido no Python. Este módulo possui uma série de funções que facilitam nossa vida ao trabalhar com data e tempo.

Data

Para criar datas através do módulo `datetime`, vamos importar o módulo e criar objetos do tipo `date`. A ideia é bem semelhante ao que vimos no último capítulo, de programação orientada a objetos.

Importaremos a classe “`date`” e criaremos um objeto que representará uma data, fornecendo ano, mês e dia, nesta ordem.

```
from datetime import date data1 = date(2016, 12, 9) print(nova_data) > 2016-12-09
```

A classe “`date`” também possui métodos úteis, como o “`today()`”, por exemplo. Este retorna a data atual local, conforme definida no seu sistema:

```
print(date.today()) > 2016-12-13
```

Também podemos usar o “`today()`” para definir uma data através do construtor da classe “`date`”, além de poder recuperar, separadamente, os valores do ano, mês e dia do objeto, através das propriedades “`year`”, “`month`” e “`day`”:

```
hoje = date.today() data2 = date(hoje.year, hoje.month, 1) print(data2) > 2016-12-01
```

Também podemos calcular a diferença entre duas datas, por exemplo, fazendo uma simples subtração entre os dois objetos e recuperando o valor conforme a medida de tempo que se deseja (dias, por exemplo). Vamos usar a função “`abs()`”, para garantir que teremos um resultado final positivo:

```
carnaval_2017 = date(2017, 2, 24) tempo_para_o_carnaval = abs(carnaval_2017 - hoje)
print(tempo_para_o_carnaval.days) > 73
```

Como era de se imaginar, temos alguns métodos bem úteis e interessantes definidos na classe “`date`”. Vamos ver alguns deles:

- `weekday()` - Retorna o dia da semana de uma data, sendo o 0 a segunda e o 7 o domingo;
- `replace()` - Retorna uma nova data, substituindo apenas os parâmetros que forem passados para este método e mantendo inalterados aqueles que não forem passados;
- `strftime()` - Retorna uma string com a data de acordo com o formato que for passado para este método. Para a informação completa com relação à formatação de data, acesse: <https://docs.python.org/3.6/library/datetime.html#strftime-strptime-behavior>.

E agora vejamos alguns exemplos destes métodos com um pouco de código:

```
print(hoje.weekday()) data2 = data2.replace(day=3) print(data2)
print(data2.strftime("%d/%m/%Y")) > 1
> 2016-12-03
> 03/12/2016
```

Tempo

Para trabalhar apenas com tempo, sem controle do dia, mês e hora, temos a classe “*time*”. Funciona de forma bem semelhante à classe “*date*”, mas seu construtor recebe apenas as horas, minutos, segundos e microsegundos. Todos têm um valor padrão de zero (0).

```
from datetime import time tempo1 = time(12, 25, 31, 1333) print(tempo1) > 12:25:31.001333
```

Para acessar suas propriedades, utilizamos o mesmo formato já visto para os objetos “*date*”, mas as propriedades são “*hour*”, “*minute*”, “*second*” e “*microsecond*” representando as horas, minutos, segundos e microsegundos do objeto “*time*”.

Para formatar a exibição de um objeto “*time*”, também usamos a função “*strftime()*”, mas com os símbolos de formatação diferentes. Vejamos um exemplo simples:

```
print(tempo1.strftime("%H horas, %M minutos e %S segundos")) > 12 horas, 25 minutos e 31 segundos
```

Data e Tempo

Quando você quer ser mais específico, representando um objeto que registre data e tempo, use a classe “datetime”. Esta classe é definida no mesmo módulo datetime que vimos anteriormente ao trabalhar apenas com datas ou tempo. E sim, o módulo tem o mesmo nome da classe.

Aqui, o construtor da classe aceita mistura o das duas classes que já vimos. Ele recebe ano, mês e dia, que são obrigatórios, além de valores para horas, minutos, segundos e microsegundos. Estes, entretanto, são opcionais. O valor padrão para cada um deles é zero (0). O formato para a criação é: datetime(ano, mes, dia, hora, minuto, segundo, microsegundo) Para retornar o momento atual, temos as funções “ now() ” e “ utcnow() ”. “ now() ” retorna a data atual do sistema, e “ utcnow() ” retorna a data atual UTC (GMT 0).

```
from datetime import datetime datatempo1 = datetime(2016, 10, 10, 18, 30, 0, 0)
print(datatempo1) datatempo2 = datetime.now() print(datatempo2) > 2016-10-10 18:30:00
> 2016-12-13 09:55:52.278336
```

Da mesma forma que com objetos “date”, você pode acessar dia, mês, segundos, de um objeto “datetime” através das propriedades “ day ”, “ month ”, “ seconds ”, e assim por diante.

```
print(datatempo2.minute) print(datatempo2.second) print(datatempo2.day) 55
52
13
```

A maioria das propriedades e muitas das funções que vimos para objetos “date” também funciona para objetos “datetime”, como a função weekday() e a possibilidade de subtrair dois objetos, por exemplo. A formatação com “strftime()” também é feita da mesma forma, e novamente, você pode acessar todas as regras de formatação de data e tempo no seguinte link: <https://docs.python.org/3.6/library/datetime.html#strftime-strptime-behavior>

Conclusão

E assim terminamos nosso capítulo sobre data e tempo. Vimos como usar o módulo “datetime” para trabalhar com data e tempo, os 3 objetos que são usados para trabalhar com data, data e tempo ou somente tempo, que são os objetos “date”, “datetime” e “time”. Vimos como criar cada um deles e acessar suas propriedades separadamente. Por fim, vimos algumas funções úteis que funcionam com os 3 tipos. No próximo capítulo veremos um pouco sobre o tratamento de erros com as Exceções.

Exceções

Exceções são uma forma de captar erros e tratá-los da maneira que você achar melhor em seu código, ao invés de simplesmente retornar uma mensagem de erro padrão do Python.

O try

O tratamento de exceções no Python é feito através de blocos “*try / except*”. A ideia já é dada pelo próprio nome. O Python vai “tentar” executar o bloco de código indentado sob a declaração “*try*”. Se ocorrer algum erro, você deve definir blocos “*except*”, de exceção, que serão rodados no caso de diferentes tipos de erro. Vejamos nosso primeiro exemplo, sem qualquer tratamento de exceções. Vamos fazer com que o Python retorne um erro, inserindo um segundo número igual a 0. Isso resultará numa divisão por 0, o que não gera um erro no Python:

```
print("Vamos dividir dois números inseridos por você\n") num1 = input("Insira o primeiro número: ") num2 = input("Insira o segundo número: ")
resultado = int(num1) / int(num2) print("O resultado é " + str(resultado))
> Vamos dividir dois números inseridos por você
> Insira o primeiro número: 12
> Insira o segundo número: 0
> Traceback (most recent call last): > File "exemplos_excecoes.py", line 5, in <module> >
resultado = int(num1) / int(num2) > ZeroDivisionError: division by zero
```

Bem, temos esta mensagem de erro do Python, que até é bem clara com relação ao erro que temos. Mas se queremos tratar este erro com mais elegância, vamos incrementar nosso código com blocos “*try / except*”:

```
print("Vamos dividir dois números inseridos por você\n") num1 = input("Insira o primeiro número: ") num2 = input("Insira o segundo número: ")
try:
    resultado = int(num1) / int(num2) print("O resultado é " + str(resultado)) except
ZeroDivisionError: print("O segundo número não pode ser zero")
> Vamos dividir dois números inseridos por você
> Insira o primeiro número: 12
> Insira o segundo número: 0
> O segundo número não pode ser zero
```

Repare que agora recebemos a mensagem de erro que definimos, sem aquelas informações adicionais do Python. Se colocarmos números corretamente, o resultado sai certo também, pois nenhum erro é detectado e o bloco de código dentro do “*try*” é executado até o fim:

```
> Vamos dividir dois números inseridos por você  
> Insira o primeiro número: 12  
> Insira o segundo número: 4  
> O resultado é 3.0
```

Podemos ter mais de uma exceção. No nosso exemplo, por exemplo, temos que pensar também que o usuário pode colocar um valor diferente de um número. Qual o erro teremos neste caso?

```
> Vamos dividir dois números inseridos por você  
> Insira o primeiro número: felipe > Insira o segundo número: 12  
> Traceback (most recent call last): > File "exemplos_excecoes.py", line 6, in <module> >  
resultado = int(num1) / int(num2) > ValueError: invalid literal for int() with base 10: 'felipe'
```

Como podemos ver, é um erro do tipo “ValueError”. Vamos incluir esta exceção em nosso código também. Ela virá logo após da primeira exceção que já definimos.

```
try:  
    resultado = int(num1) / int(num2) print("O resultado é " + str(resultado)) except  
ZeroDivisionError: print("O segundo número não pode ser zero") except ValueError:  
    print("Você deve inserir dois números")
```

Agora, vamos fazer o teste desta nova exceção:

```
> Vamos dividir dois números inseridos por você  
> Insira o primeiro número: felipe > Insira o segundo número: 12  
> Você deve inserir dois números
```

Funcionando perfeitamente. Você pode utilizar quantas exceções quiser, pegando todos os tipos de erro que podem ser cometidos pelo usuário em cada caso. E se eu quiser pegar todos os erros? Basta usar somente o except, sem qualquer definição do tipo de erro. Mas algumas literaturas não consideram isto como uma boa prática, pois ele não identifica a raiz do problema e também não permite passar uma informação mais completa e precisa para o usuário. Mas de qualquer forma, ficaria assim:

```
try:  
    resultado = int(num1) / int(num2) print("O resultado é " + str(resultado)) except:  
    print("Uma das entradas é inválida. Favor inserir dois números, sendo o segundo diferente  
    que zero")
```

```
> Vamos dividir dois números inseridos por você >
> Insira o primeiro número: 12
> Insira o segundo número: 0
> Uma das entradas é inválida. Favor inserir dois números, sendo o segundo diferente que > zero

> Vamos dividir dois números inseridos por você >
> Insira o primeiro número: felipe > Insira o segundo número: 1
> Uma das entradas é inválida. Favor inserir dois números, sendo o segundo diferente que > zero
```

No exemplo acima, rodamos o código duas vezes, uma para cada erro que já vimos. O tratamento para ambos é o mesmo, com a mesma mensagem exibida ao usuário.

Vou manter o código anterior, com o tratamento das exceções pelo tipo de erro, e comentar o código acima para prosseguir.

Para ver todos os tipos de exceção, acesse o link abaixo:
<https://docs.python.org/3/library/exceptions.html>

else e finally

Temos mais duas estruturas que podemos usar com as exceções no Python, que são o “else” e o “finally”. O “else” é um bloco final, que roda após o código no bloco “try” rodar sem qualquer erro. Vejamos:

```
try:
    resultado = int(num1) / int(num2) print("O resultado é " + str(resultado)) except
ZeroDivisionError: print("O segundo número não pode ser zero") except ValueError:
    print("Você deve inserir dois números") else:
    print("Divisão feita!")
```

Utilizando os números adequados, veremos a mensagem final ser impressa, conforme está definido no bloco “else”. Caso caia na exceção, o “else” não é executado:

```
> Vamos dividir dois números inseridos por você >
> Insira o primeiro número: 12
> Insira o segundo número: 3
> O resultado é 4.0
> Divisão feita!

> Vamos dividir dois números inseridos por você >
> Insira o primeiro número: 12
> Insira o segundo número: 0
> O segundo número não pode ser zero
```

Repare que o código neste bloco “ else ” só é executado quando não incorremos em qualquer exceção, ou seja, o bloco “ try ” é executado até o final sem qualquer tipo de erro.

Já o “ finally ” define um bloco que sempre será executado, independente de ocorrer um erro ou do bloco definido no “ try ” ter sido executado sem erros.

```
try:
    resultado = int(num1) / int(num2) print("O resultado é " + str(resultado)) except
ZeroDivisionError: print("O segundo número não pode ser zero") except ValueError:
    print("Você deve inserir dois números") finally:
```

```
print("Programa concluído")
```

Primeiro, sem entrar em uma exceção:

```
> Vamos dividir dois números inseridos por você >  
> Insira o primeiro número: 12  
> Insira o segundo número: 3  
> O resultado é 4.0  
> Programa concluído
```

E agora, entrando em uma exceção:

```
> Vamos dividir dois números inseridos por você >  
> Insira o primeiro número: 12  
> Insira o segundo número: 0  
> O segundo número não pode ser zero > Programa concluído
```

Não importa, o código dentro do bloco “*finally*” sempre será executado. E podemos usar os dois juntos, tendo um bloco “*else*” e um bloco “*finally*” após a definição das suas condições.

Chamando Exceções

Você também pode chamar exceções em seu programa, através da declaração “*raise*”. Vamos chamar uma exceção em um bloco “*try*” se o nome inserido for “*Felipe*”:

```
nome = input("Qual o seu nome? ") try:  
if nome == "Felipe": raise NameError("Não gostei do seu nome") print("Olá, %s" %  
nome) except NameError: print("O programa não gostou do seu nome")
```

Se inserimos um nome qualquer, o bloco “*try*” chega até seu final e o bloco “*except*” não é executado. Desta forma, vemos apenas a mensagem “Olá, nome”:

```
> Qual o seu nome? José > Olá, José
```

Porém, se inserirmos “Felipe” como nome, uma exceção será chamada, conforme está definida dentro do bloco “*if*” pela declaração “*raise*”, e o bloco “*except*” será chamado. Não veremos, então, o “*print*” que está definido na última linha do bloco “*try*”, e sim a mensagem definida no bloco “*except NameError*”:

```
> Qual o seu nome? Felipe > O programa não gostou do seu nome
```

Conclusão

Concluimos assim o capítulo sobre exceções e tratamento de erros. Assim, seu programa pode passar informações ao usuário de forma mais elegante, prevendo os possíveis erros cometidos pelo usuário e passando mensagens informativas para o usuário caso ocorram.

Arquivos

Neste capítulo, veremos como é possível usar o Python para trabalhar com arquivos. O Python tem funções para ler, gerar e escrever em arquivos, e veremos cada uma delas.

A função open()

Esta é a função mais primária para se trabalhar com arquivos. Ela serve, como o nome já denuncia, para “abrir” arquivos. Ela recebe dois parâmetros, o nome do arquivo e o modo de abertura. O modo indica o que você pode e não pode fazer com o arquivo. A tabela abaixo lista os principais modos:

Modo	Descrição
r	Abre o arquivo apenas para leitura
w	Abre o arquivo apenas para escrita. Se existe um arquivo com o nome especificado, este será apagado e um novo será criado
a	Abre o arquivo para acrescentar a ele. Tudo que for escrito no arquivo é adicionado ao final dele
r+	Abre o arquivo tanto para leitura quanto para escrita

Estes são os métodos mais utilizados, apesar de termos outros métodos, cuja lista completa pode ser encontrada no seguinte link: <https://docs.python.org/3.6/library/functions.html#open>

Vamos então abrir um arquivo de texto. Criei um arquivo txt com alguns nomes de pessoas aleatórios, se quiser pode baixar aqui, ou, se não quiser, pode criar o seu mesmo: <https://drive.google.com/file/d/0B46dernCkjj7YjFOSVBSWlhUa2s/view?usp=sharing>

Vamos então criar nosso arquivo .py com o código para ler este arquivo:

```
file = open("nomes.txt", "r") print(file.read())
> Felipe > Carolina > Joana > José > Alfredo > Afonso > Júlio > Ana > Joelma > Flash Gordon
> Vanderlei > Maria Claudia
```

Passando um número inteiro para a função “*read()*”, nós podemos definir a quantidade de caracteres a serem retornados na leitura. Atentem ao

detalhe que o objeto “ *file* ” guarda a uma determinada posição dentro do arquivo (pense como um ponteiro do Microsoft Word, que indica onde você está escrevendo / lendo). Ao abrir o arquivo, esta posição é 0, ou seja, o primeiro caractere do arquivo. Após usar a função “ *read()* ”, o ponteiro se move para o último caractere. Então, após a primeira leitura, vamos abrir o arquivo novamente para fazer a leitura a partir do começo:

```
file = open("nomes.txt", "r") print(file.read(10))  
> Felipe > Car
```

A função “ *readline()* ” lê uma única linha do arquivo. Caso você siga repetindo a função, a mesma vai lendo as linhas seguintes do arquivo:

```
file = open("nomes.txt", "r") print(file.readline()) print(file.readline()) print(file.readline())  
> Felipe >  
> Carolina > > Joana
```

Bem semelhante, a função *readlines()* retorna uma lista, sendo cada item uma linha do arquivo lido:

```
file = open("nomes.txt", "r") print(file.readlines())  
> ['Felipe\n', 'Carolina\n', 'Joana\n', 'José\n', 'Alfredo\n', 'Afonso\n', 'Júlio\n', 'Ana\n', 'Joelma\n',  
'Flash Gordon\n', 'Vanderlei\n', 'Maria Claudia']
```

Desta forma, você pode iterar sobre as linhas e processá-las da forma que desejar.

Escrevendo em um arquivo

Vamos agora escrever em um arquivo, usando para isso o modo “w” na função “*open()*”, conforme vimos mais acima. Depois, usamos a função “*write()*” para escrever no arquivo aberto. Além disso, também é importante lembrar de fechar o arquivo com a função “*close()*”. O Python faz isso para nós ao final de um programa, mas é uma boa prática fechar o arquivo após usá-lo. Vamos escrever duas linhas de texto qualquer:

```
file = open("novo_arquivo.txt", "w") file.write("Testando, testando!\n") file.write("Mais uma linha para testar") file.close()
```

Agora, verifique na mesma pasta onde está salvo o seu arquivo .py, e, se tudo correu bem, deve existir um arquivo com o nome “novo_arquivo.txt” e com o mesmo conteúdo que escrevemos dentro das funções “*write()*”.

Agora, vamos escrever outro texto em um arquivo de mesmo nome, apenas para confirmar que a abertura do arquivo neste modo sobrescreve o arquivo definido:

```
file = open("novo_arquivo.txt", "w") file.write("Novo texto, mesmo arquivo") file.close()
```

E certamente, ao abrir o arquivo “novo_arquivo.txt”, encontraremos apenas o conteúdo que definimos na função “*write()*” nesta segunda vez em que abrimos o arquivo.

Adicionando a um arquivo

Vamos agora adicionar texto a um arquivo já existente. Para isso, vamos abrir o arquivo no modo “a”. Podemos usar o arquivo que criamos no último exemplo, “novo_arquivo.txt”. A função neste modo é a mesma usada para novos arquivos, “*write()*”:

```
file = open("novo_arquivo.txt", "a") file.write("\nTexto adicionado ao arquivo com o modo a") file.close()
```

Agora, ao abrir “novo_arquivo.txt”, você verá que a linha acima foi adicionada no final do mesmo.

Abrindo no modo r+

O modo “r+” permite tanto a leitura quanto a escrita em arquivos. A posição inicial ao abrir o arquivo neste modo é 0, de forma que qualquer coisa que seja escrita sem que se mova a posição substituirá o que está no começo.

Você pode alterar a posição através da função “ *seek()* ”, fornecendo a posição para a qual você deseja alterar:

```
file = open("novo_arquivo.txt", "r+") file.write("Testando r+") file.seek(10)
file.write("Novo teste do r+") file.close()
```

Veja agora como o conteúdo do arquivo está meio misturado. Primeiro, abrimos no modo “r+” e já escrevemos, substituindo o conteúdo no início do arquivo. Depois, usamos a função “ *seek()* ” para mudar nossa posição para 15. E então escrevemos novamente, substituindo mais uma parte do conteúdo que já existia no arquivo. Podemos usar o “ *seek()* ” novamente para retornar para a posição 0 e então ler o arquivo.

```
file.seek(0) print(file.read()) file.close()
> Testando r+ mesNovo teste do r+ adicionado ao arquivo com o modo a
```

Conclusão

E assim terminamos o capítulo sobre arquivos. Vimos como criar, ler e manipular arquivos com o Python, escrevendo e adicionando conteúdo a eles. Como a função `open()` e o modo de abertura definem o que pode e o que não pode ser feito com o arquivo, e como navegar em um arquivo aberto com a função “`seek()`”.

Palavras Finais

E assim, esse livro chega ao fim. Espero ter conseguido passar um pouco de conteúdo legal para vocês que chegaram até aqui. Acho que com o que foi passado, dá para se ter uma boa noção da linguagem Python.

Caso queiram entrar em contato, basta ver como em:
<http://felipegalvao.com.br/>

E agora?

Bem, agora você pode alçar vôos mais altos. O que te interessa? Você pode pegar o que aprendeu de Python e aplicar em estudos posteriores. Algumas sugestões baseado no que o Python pode fazer:

- Desenvolvimento Web
 - Django - <https://www.djangoproject.com/>
 - Flask - <http://flask.pocoo.org/>
 - Pyramid - <https://trypyramid.com/>
- Análise de Dados, Ciência de Dados e Computação Científica
 - Propaganda sem vergonha - No meu blog tem uma série de posts sobre como começar com análise de dados usando Python: <http://felipegalvao.com.br/blog/ciencia-de-dados-com-python-basico/>
 - pandas (análise de dados) - <http://pandas.pydata.org/>
 - Numpy (computação científica) - <http://www.numpy.org/>
 - SciPy (programação científica) - <https://www.scipy.org/>
 - scikit-learn (Machine Learning) - <http://scikit-learn.org/stable/>
 - BioPython (programação para biologia) - <https://biopython.org/>
- Programação para Desktop
 - Camelot - <http://www.python-camelot.com/>
 - PySide - <http://wiki.qt.io/PySide>
 - tkinter - <https://docs.python.org/3/library/tkinter.html>
- Programação multiplataforma
 - Kivy - <https://kivy.org/#home>
- Desenvolvimento de Jogos
 - Pygame - <http://pygame.org/hifi.html>
 - Panda3D - <http://www.panda3d.org/>

Pesquise o que mais te interessa, material não falta pela Internet. Python tem boas bibliotecas para praticamente tudo.