

A large, faint, light blue Python logo watermark is centered in the background of the slide. The logo consists of two interlocking snakes, one facing left and one facing right, forming a circular shape.

PYTHON

SUMÁRIO

03
Introdução à programação

40
Arquivos

09
Programação em Python:
Tipos de dados e Operadores

47
Estruturas de Dados II:
Tuplas e Dicionários

17
Controle de Fluxo

53
Programação Orientada a Objetos

21
Estruturas de dados: Listas

60
Aplicações GUI

29
Funções

68
Requisições e APIs

33
Tipos de dados:
Strings

75
Introdução a HTML
e Web Scraping

The background of the entire page is a dark blue field filled with a complex, light blue circuit-like pattern. This pattern consists of numerous thin, interconnected lines that branch out and connect to small circular nodes, resembling a printed circuit board or a neural network diagram. The lines and nodes are more densely packed on the left side and become sparser towards the right.

CAPÍTULO

01

INTRODUÇÃO À
PROGRAMAÇÃO

1.1 O que é um programa

Antes de começar a programar, precisamos entender o que é um programa. Um programa é uma sequência de instruções a serem seguidas para resolver um problema ou alcançar um determinado objetivo. Pode-se dizer que o programa é como a receita de um bolo, por exemplo.

Essas instruções — ou tarefas — que serão passadas ao computador podem tomar várias formas, podendo ser instruções matemáticas, como adicionar, subtrair, multiplicar e dividir números; operações sobre textos, como buscar e/ou substituir partes de um texto e até processamentos gráficos, como tela, exibição de uma imagem ou vídeo; entre outras inúmeras possibilidades.

“When you’re programming, you’re teaching - probably the stupidest thing ever — a computer — how to do something.”

Gabe Newell

compostos por um elemento de processamento (um dispositivo de entrada e/ou saída de informações), uma central de processamento (CPU) e alguma estrutura capaz de armazenar informações: a memória do computador.

Resumidamente, um computador é uma máquina capaz de receber um ou mais dados de entrada, armazená-los de alguma maneira, e então processar esta(s) entrada(s) da maneira como foi programado para fazer. Nós, (futuros) programadores somos responsáveis pela etapa de processamento desses dados, afinal, é neste momento que o computador recorre às instruções que escrevemos para que ele saiba exatamente o que fazer com os dados que recebeu.

Como um exercício para melhorar o entendimento da arquitetura de um computador, você pode pensar nos eletrodomésticos da sua casa: a máquina de lavar, a geladeira, o ar-condicionado. Pense em como você ou o ambiente ao seu redor passa informações para estas máquinas e a maneira como elas reagem a estas diferentes informações.


1.2 Como o computador funciona

Vamos tentar nos retratar um pouco após a frase do criador da Valve, Gabe Newell, sobre os computadores (para não começarmos já em pé de guerra com eles!), e entender melhor como essas estúpidas maravilhas funcionam, tudo bem?

Os computadores, desde um forno microondas convencional até os laptops e smartphones de uso pessoal, são basicamente

1.3 Linguagens de programação

As linguagens de programação são o nosso meio de passar essas instruções ao computador. Existem literalmente **centenas** de linguagens e por isso nós as separamos de acordo com alguns critérios a fim de facilitar um pouco a



nossa busca pela linguagem ideal para resolver um problema dado. Veja como os nossos critérios foram pensados:

Linguagem de Alto nível e Linguagens Baixo Nível

Linguagens de alto nível são aquelas com alto nível de abstração, ou seja, se aproximam mais de como nós, humanos, entendemos o mundo. As linguagens de programação mais famosas estão nesta categoria, como por exemplo: Python, Java, JavaScript, C++, entre muitas outras.

As linguagens de baixo nível, por sua vez, são aquelas que fogem de como nós vemos o mundo ao nosso redor sendo mais próximas de como o nosso computador o “vê”, de como os seus pequenos e inúmeros circuitos elétricos processam cada informação. Estas linguagens são as “mães” das linguagens modernas de alto nível, isto é, foi a partir das linguagens de baixo nível que foram criadas as linguagens de alto nível. O exemplo clássico é a linguagem Assembly x86, utilizada pela arquitetura da grande maioria dos computadores pessoais do mercado.

Linguagens de Uso Geral e Linguagens de Domínio Específico

Dizemos que uma linguagem de programação é de uso geral se ela pode ser usada para resolver problemas de diversas naturezas, como a criação de um aplicativo para web, mobile, sistemas desktop, etc.

Já as linguagens de domínio específico são aquelas criadas e utilizadas para uma aplicação específica, como é o caso do SQL, por exemplo, que é usada para consultas

em bancos de dados.

Linguagens Interpretadas e Linguagens Compiladas


As linguagens interpretadas são aquelas que trabalham em conjunto com um interpretador, um software que interpreta o código escrito linha a linha e executa os comandos devidos direto na CPU. Não é possível executar um programa feito em linguagem interpretada sem ser em conjunto com o interpretado.

Já as linguagens compiladas, por sua vez, se utilizam de um compilador, um software capaz de gerar um arquivo executável, em linguagem de máquina, para ser então executado pela CPU de forma independente. Uma vez compilado, não há mais necessidade do compilador para executar o programa.

1.4 Componentes de uma linguagem

Quando comparamos linguagens de programação, — e os fanáticos programadores fazem isso com mais afinco do que defendem seus times de futebol, é normal que comparemos os dois componentes básicos de uma linguagem de programação, em especial o quão fáceis ou robustos estes são e se desenvolvem.

O primeiro deles, e provavelmente o mais corriqueiro na hora da comparação é a **sintaxe**. Ela define como o código daquela linguagem deve ser redigido, se deve obedecer a muitas regras de escrita ou não. É quase como comparar regras gramaticais de diferentes línguas.



O segundo é a **semântica**. Ela é o componente que entra em cena após a sintaxe e refere-se ao sentido para o computador dos comandos que se escreve. Comparar semântica é comparar, entre duas linguagens, em qual é mais fácil realizar determinada tarefa.

1.5 Introdução ao Python

A linguagem Python é open-source¹ e está disponível para download no site <https://www.python.org>. Ao acessar o site, basta você escolher o sistema operacional do seu computador e iniciar o download.

Obs: Caso o seu SO seja Linux, você provavelmente já possui o Python instalado. Abra o terminal, digite `python3 --version` e faça o teste. Caso você já tenha, o terminal retornará a versão da linguagem, mas caso ele não reconheça o comando, você pode instalá-lo facilmente digitando `sudo apt-get install python3`.

Em nosso curso, utilizaremos a versão 3.x do Python. Existem algumas diferenças entre a sintaxe do Python 2.x e 3.x e focaremos nosso estudo na última versão (3.x), já que naturalmente uma versão mais nova nos dá mais recursos.

Após baixar o Python 3.x, instale-o como qualquer outro programa. Quando perguntado, selecione a opção de "incluir o Python no PATH do sistema", isto nos permitirá usá-lo com mais facilidade no terminal.

comando: `python` (`python3` no Mac). Abra uma janela do CMD (no Windows) ou seu terminal, caso esteja usando Linux, e digite `python` + `enter`, para acionar o terminal interativo, também chamado de shell.

Toda vez que executamos o Python de maneira simples no CMD, vemos uma mensagem com a versão do Python, alguns comandos para mais informações e um prompt com três "maior que", a partir daí tudo que escrevermos será interpretado como código Python.

1.6 Python no terminal (CMD) do computador

Depois de instalada a linguagem, o computador passa a ter um novo

É tradição toda vez que aprendemos uma nova linguagem, começarmos fazendo com que o computador diga "Hello, world!". A tradição começou em 1978 no livro "The C Programming Language" e é seguida até hoje. Não vai ser você quem vai querer quebrar, né?! Para imprimir algo na tela usando

¹ além de usar sem custo, qualquer um pode alterar a linguagem para suas necessidades sem pagar nada para ninguém

Python, utilizamos a função **print()**, passando entre os parênteses, o valor que desejamos imprimir. Então, para começar bem o nosso estudo e garantir que os deuses da computação não irão se zangar, escreva no seu terminal `print("Hello, world!")` e tecle Enter. Agora sim podemos começar.

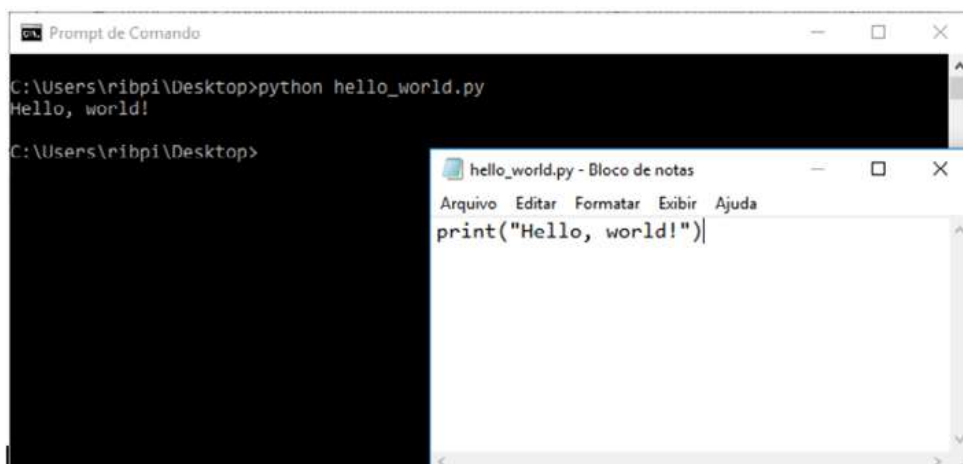
```
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello, world")
Hello, world
>>> █
```

1.7 Editores de texto, IDEs, e compiladores

O uso do terminal para execução de código Python, é mais comum para testar uma funcionalidade, ou relembrar o uso de uma função ou uma propriedade da linguagem. O que executamos no terminal não fica armazenado para reuso em local nenhum. Quando escrevemos um programa robusto, estruturado como um produto final, armazenaremos o nosso código Python em um (ou mais) arquivo de texto.

Você já deve estar acostumado com alguns formatos de arquivos mais comuns, como .txt ou .docx para arquivos de texto, .png, .jpeg, .gif para imagens, entre outros formatos que armazenam áudio, vídeo, e outros formatos de dados e conteúdo. Um arquivo com código Python deve ser salvo com a extensão .py, assim você poderá executá-lo direto do seu terminal com o comando **python** nome_do_arquivo.py.

Esses arquivos podem ser criados e modificados em qualquer editor de texto, pode ser o mais simples que o seu sistema operacional fornece. Basta, ao salvar, mudar a extensão padrão (normalmente .txt) para **.py**.



No entanto, à medida que o nosso programa aumenta em complexidade, esses editores mais simples deixam de atender bem às nossas necessidades. Existem muitos editores de texto, focados em edição de código, disponíveis para download. Um dos mais poderosos e famosos no mercado é o [Sublime Text 3](#). Este editor tem suporte para várias linguagens de programação e possui funcionalidades que te ajudarão muito no desenvolvimento do seu código, como autocompletação,

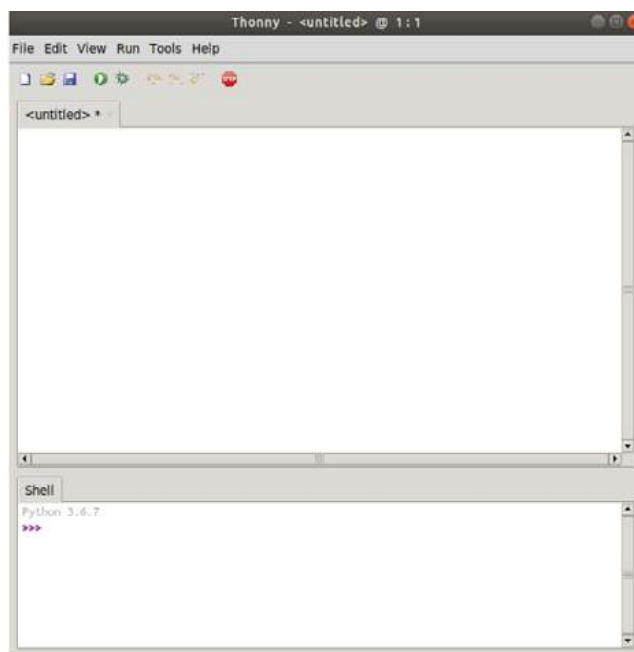
coloração da sintaxe do código, entre outras. Caso decida utilizar o Sublime Text 3, você pode consultar [aqui](#) um guia de configuração do editor para a linguagem Python, ampliando ainda mais os seus poderes de edição de código.

É muito comum a confusão entre editores de texto e IDEs (Integrated Development Environment) ou ambientes de desenvolvimento integrado. Os IDEs são também editores de texto, mas normalmente eles oferecem funções “extras”, além de ajudar a redigir o seu código, IDEs costumam ter mecanismos internos para debugging, execução do código e algumas vezes até geração automática de código. Quando dizemos que os IDEs são capazes de executar o código, dizemos que possui um **compilador ou interpretador** interno. Os compiladores e interpretadores são independentes de um editor de texto ou um IDE, você tem trabalhado com um interpretador desde o nosso “Hello, World!”. Porém, é comum que exista um compilador ou interpretador integrado já no IDE, para ajudar no desenvolvimento do seu programa, que é o seu propósito final. Como o Python é uma linguagem interpretada, os IDEs para Python provém um interpretador.

Neste curso, usaremos o **Thonny**, um IDE totalmente gratuito e open-source. Você pode inclusive acessar o código-fonte do Thonny e até contribuir para o seu desenvolvimento, se quiser..

O download e a instalação do Thonny são muito simples, e estão disponíveis para Windows, Linux e Mac. Basta acessar a página do IDE, escolher seu sistema operacional e

iniciar o download. Se estiver utilizando o sistema Linux, terá a opção de baixar e instalar o Thonny seguindo algumas instruções fornecidas no próprio site do IDE. Se preferir, você também poderá utilizar um assistente de aplicativos disponibilizado, que pode ser imprescindível para a instalação, dependendo da distribuição que está utilizando na sua máquina.



A interface do Thonny é bem intuitiva, tendo basicamente duas partes principais. Uma delas é para edição de arquivos (parte superior) e a segunda é o shell, assim como nós utilizamos no terminal. As partes são independentes, embora toda vez que você peça a execução do seu arquivo, clicando no botão Run ou apenas teclando F5, o resultado aparecerá no shell.

² Também conhecido como depuração, é o processo de correção de um programa com problemas desde a análise do código para buscar erros diversos até seu efetivo reparo para que funcione como o esperado.

The background of the entire page is a dark blue field with a complex, light blue circuit-like pattern. This pattern consists of numerous thin, interconnected lines and small circular nodes, resembling a printed circuit board or a digital network. The lines vary in thickness and orientation, creating a sense of depth and movement across the frame.

CAPÍTULO

02

TIPOS DE DADOS E
OPERADORES

2.1

Operadores Aritméticos em Python

Operadores aritméticos são aqueles que normalmente são utilizados para realizar alguns cálculos entre números, como adições, subtrações, multiplicações e divisões.. Todas estas simples operações, e outras mais, são nativamente tratadas no sistema Python. Vamos então começar os nossos estudos por elas.

Utilizando a função **print()** que já aprendemos para inaugurar nosso aprendizado de Python, faça um teste no terminal e tente imprimir uma soma simples entre dois números inteiros.

Os operadores nativos de aritmética do Python para as operações são: (+) para adições, (-) para subtrações, (*) para multiplicações e (/) para divisões. A partir desses operadores, podemos fazer cálculos entre quaisquer números.

Além destas operações mais básicas, o Python também lida nativamente com operações de potência, com o operador (**); módulo, com o operador (%); e divisões inteiras, com o operador (//).

a ** b -> Calcula o valor da base **a** elevada ao expoente **b**

a % b -> Retorna o resto da divisão de **a** por **b**

a // b -> Divide **a** por **b**, retornando apenas a parte inteira do resultado da divisão

As operações em Python seguem as mesmas regras de prioridades que você já conhece da matemática,

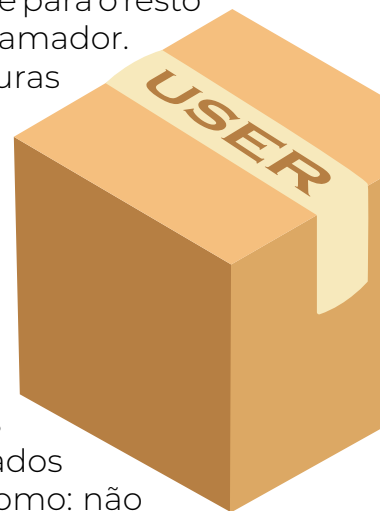
tente combinar algumas operações e perceba como a ordem de prioridades se mantém. Os parênteses também são respeitados como prioridade, da mesma forma como você já deve ter estudado., Esta relação só não se estende ao uso de colchetes e chaves, que por vezes utilizamos no papel.

2.2

Variáveis

O uso de variáveis nos acompanhará em todo o resto do curso, e para o resto da sua vida como programador.

As variáveis são estruturas para armazenamento de dados em memória capazes de guardar um determinado valor que, por sua vez, como o próprio nome sugere, pode variar. Toda variável deve possuir um nome, mas existem alguns cuidados a serem tomados na hora de nomeá-las, como: não iniciar os nomes com números, não utilizar caracteres especiais (/, %, &, ~, ^, ...), e não utilizar espaços.



```
variables.py *  
"Valid variables:"  
number = 3  
_number = 4  
number = 5 + 8  
  
"Invalid variables:"  
number = 2**2  
num%ool = 6 // 2  
numbêr = 7 % 2  
my number = 95
```

Obs: o Python é uma linguagem case-sensitive, isto é, diferencia letras maiúsculas de minúsculas, por exemplo, as variáveis de nome Number, number e NUMBER são totalmente independentes e podem guardar diferentes valores sem o menor problema.

Para armazenar um valor em uma variável, usamos o sinal de igual (=), que é chamado operador de atribuição em Python, utilizado para atribuir um valor a uma variável. Podemos também fazer múltiplas atribuições em apenas uma linha, como demonstrado abaixo:

```
Shell
>>> x, y, z = 1, 2, 3
>>> print(x)
1
>>> print(y)
2
>>> print(z)
3
```

Além do operador de atribuição, existem também os operadores de incremento (+=) e de decremento (-=).

Usamos o operador de incremento quando o valor que desejamos armazenar na variável corresponde ao seu valor anterior, acrescido de um “incremento”. Exemplo:

```
Shell
>>> letscore_students = 200
>>> # two more students register for the Python course
>>> letscore_students += 2
>>> print(letscore_students)
202
```

O operador de decremento, por sua vez, funciona da maneira oposta,

sendo utilizado quando o valor que a variável deva armazenar é o valor que ela já possuía, menos um “decremento”. Como mostramos a seguir:

```
Shell
>>> letscore_students = 200
>>> # five students finish the course
>>> letscore_students -= 5
>>> print(letscore_students)
195
```

O conceito mostrado também se estende para operadores de multiplicação (*=) e divisão (/=).

Diferentes linguagens possuem diferentes especificações sobre como suas variáveis devem ser nomeadas, o jeito “pythônico” de nomear variáveis estabelece que as variáveis devem ser compostas por letras minúsculas, e em caso de nomes compostos por mais de uma palavra, separando-os por um (_), como por exemplo:

```
>>> character_name = "Darth Vader"
```

Obs: Embora recomendadas, o não uso destas regras não trará erros para o programa, estas especificações são conhecidas apenas como boas práticas da programação, e cada linguagem possui as suas. As boas práticas de programação em Python estão descritas em um documento chamado PEP8, disponível no site oficial da linguagem.

Essas especificações melhoram a legibilidade do nosso código, facilitando o entendimento de outras pessoas ao ler e até nosso próprio entendimento, quando revisitarmos o código em algumas semanas ou meses.

Um recurso interessante do Python, presente na maioria das linguagens de programação é a possibilidade de fazer comentários. Comentários são textos breves, utilizados para fazer uma breve descrição de uma linha ou bloco de código que o interpretador ignora, isto é, que não influenciam de maneira nenhuma no código. Os comentários podem ajudar muito na legibilidade de algumas linhas de código, que mesmo que sejam bem escritas, por fazerem operações muito complexas, são difíceis de compreender a primeira vista.

Os comentários são iniciados por um sinal de tralha, também conhecido por hash ou cerquilha (#). Embora não seja boa prática, pode ser que se observe alguns comentários sendo feitos simplesmente entre aspas. Neste caso, a linha de “comentário” é sim interpretada durante a execução do programa. O interpretador considera este caso como valor textual sem qualquer operação sobre a mesma. Ressalta-se que a utilização de aspas para comentários **não é recomendada**.

“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.”

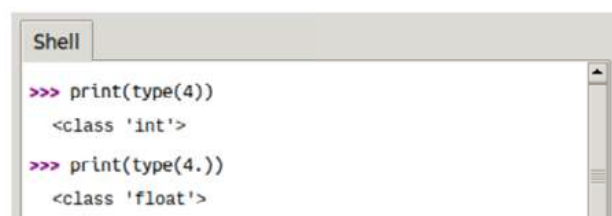
Martin Fowler

2.3 Tipos de dados: Integers e Floats

Quando estamos trabalhando com valores numéricos em Python, existem dois tipos básicos com que lidamos, os integers (int), que guardam números

inteiros – que grande surpresa, não? – e os floats que guardam números reais representados pela normalização de **ponto flutuante**, origem do termo **float**.

Para consultar o tipo de um valor (diretamente ou a partir de uma variável) podemos usar a função **type()**. O Python diferencia os tipos **int** e **float** simplesmente pela presença do ponto flutuante. Faça o teste no seu terminal, execute os comandos abaixo:

A screenshot of a terminal window titled "Shell". It shows two commands and their outputs. The first command is `>>> print(type(4))` and the output is `<class 'int'>`. The second command is `>>> print(type(4.))` and the output is `<class 'float'>`.

Embora pareça que o Python lide muito bem com números reais (e na verdade lida mesmo) é importante ressaltar que todo cálculo feito em Python – ou em qualquer outra linguagem de programação – é realizado utilizando representação binária que contém imprecisões. Este fator pode causar algumas inconsistências. Para ilustrar, vamos utilizar o Python para uma simples operação de adição, execute o seguinte em seu terminal:

```
>>> print(0.1 + 0.1 + 0.1)
```

Se você tentou fazer “de cabeça” antes de usar o terminal, você provavelmente chegou ao resultado 0.3, no entanto, ao executar este cálculo, você vai perceber que o terminal imprimirá o valor 0.30000000000000004, que se refere à representação de números reais normalizada em ponto flutuante.

Podemos converter os valores de **int** para **float** e vice-versa usando o mecanismo que chamamos de **casting**. Essas conversões são comuns em outras

linguagens e são usadas sempre que precisamos converter tipos ou aplicações semelhantes (veremos melhor este caso no tópico sobre **strings**). Para converter um valor float em inteiro, basta utilizar **int(numero_real)**, para fazer o contrário, use **float(numero_inteiro)**. Faça o teste!

2.4 Tipos de dados: Strings

Strings são como chamamos em Python os tipos de dados que guardam um valor textual, assim como o primeiro que utilizamos ("Hello, world!"). Trazendo este exemplo de volta, poderíamos ter feito algo como:



```
Shell
>>> greetings = "Hello, World!"
>>> print(greetings)
Hello, World!
```

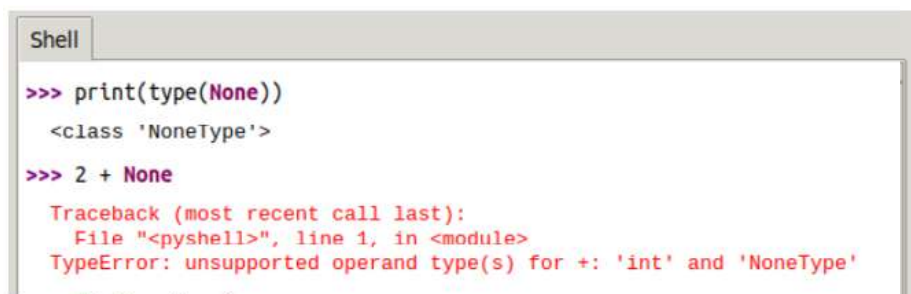
O resultado seria o mesmo. O valor de uma **string** pode ser delimitado por aspas duplas – como no exemplo acima, ou por aspas simples. Logo, o exemplo acima também poderia ser escrito da seguinte forma:



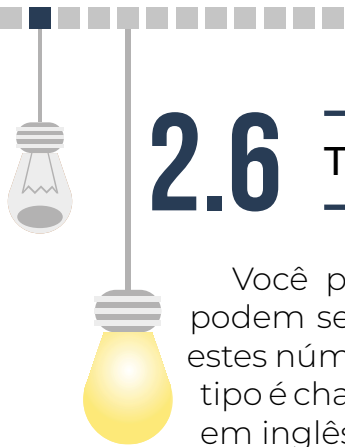
```
Shell
>>> greetings = 'Hello, World!'
>>> print(greetings)
Hello, World!
```

2.5 Tipos de dados: NoneType

Há um tipo especial de valor em Python, diferente dos que vimos até agora. Este tipo de dado possui apenas um valor, o valor **None**. O valor **None** é uma forma de indicar que uma variável não deve possuir qualquer conteúdo, em outras palavras, que seu valor é nulo. Este tipo não possui nenhum método ou atributo especial e tampouco pode estar numa operação com inteiros ou **strings**.



```
Shell
>>> print(type(None))
<class 'NoneType'>
>>> 2 + None
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```



2.6 Tipos de dados: Booleanos

Você provavelmente já ouviu falar em números binários (aqueles que só podem ser 0 ou 1). Em Python, existe outro tipo de dados que se assemelha a estes números por poder ter também apenas dois tipos distintos de valores.. Este tipo é chamado de booleano, ou **bool**, que só pode ter os valores **True** (verdadeiro em inglês) e **False** (falso em inglês).

2.7 Input/Output

Desde o começo do nosso curso, nas nossas primeiras interações diretas com o Python, estamos utilizando uma função de saída de dados, que é a função **print()**. Com ela, podemos mostrar para o usuário do nosso programa mensagens, valores de variáveis ou resultados de operações feitas internamente.

Agora é a hora de conhecer uma função que nos permite “conversar” com o usuário, além de apenas mostrar valores e mensagens. Para isso, na maioria das vezes, precisamos receber uma entrada do usuário, a partir da qual o programa pode fazer alguma operação. A forma mais simples de utilização desse recurso de entrada de dados do usuário é a função **input()**.

A função **input()** recebe um parâmetro de mensagem para saber como o usuário realizará o pedido de uma determinada informação, e como o programa deve retornar esta entrada. Para ilustrar, teste o exemplo abaixo no seu terminal:

```
strings.py x
name = input("What's your name?")
print("Hello,", name)
```

É importante ressaltar que o retorno da função **input()** sempre será um tipo **string**, logo, caso a informação que você queira do usuário vá ser submetida a alguma operação aritmética, você deve utilizar as funções de **casting** para transformar o retorno da função. Por exemplo:

```
strings.py x
age = int(input("How old are you?"))
age += 10
print("In ten years, you'll be", age, "years old. If you don't die of course")
```

2.8 Operadores de comparação

Você já foi apresentado aos valores booleanos no tópico de Tipos de Dados, esses valores são naturalmente resultado de alguma operação de comparação entre valores, sejam estes numéricos ou textuais. Os operadores de comparação em Python são aqueles que você já conhece da matemática, mas talvez com uma representação um pouco diferente:

| | |
|----|------------------|
| > | maior que |
| < | menor que |
| >= | maior ou igual a |
| <= | menor ou igual a |
| == | igual a |
| != | diferente de |

Utilizando esses operadores, você pode comparar tanto textos quanto números. Execute as linhas abaixo no seu terminal e observe o resultado.

```
Shell
>>> print(3 < 5)
True
>>> print(3 > 5)
False
>>> print(3 == 5)
False
>>> print(3 != 5)
True
```

```
Shell
>>> print('a' < 'b')
True
>>> print('a' < 'A')
False
>>> print('a' == 'A')
False
>>> print('Hello, world' != 'Hello, World')
True
```

Obs: As comparações de “menor que” e “maior que” sobre textos ou caracteres, seguem a convenção da tabela ASCII, uma tabela que relaciona caracteres com números inteiros, permitindo assim este tipo de comparação. A tabela ASCII também é muito importante no momento de interpretar representações binárias das letras para a letra em nosso alfabeto.

2.9 Operadores lógicos

Os operadores lógicos são os usados para trabalhar diretamente com valores booleanos de uma ou mais expressões. Existem três operadores lógicos em Python: **and**, **or** e **not**. O uso desses operadores é bem intuitivo, os dois primeiros são usados para compor uma condição final que analisa duas comparações.

O operador **and** retornará **True** apenas quando ambas as comparações analisadas forem verdadeiras e **False** para qualquer outro caso. Por exemplo:

```
Shell
>>> print(3 > 5 and 3 != 5)
False
>>> print(3 > 5 and 3 != 5)
False
>>> print(3 < 5 and 3 == 5)
False
>>> print(3 < 5 and 3 != 5)
True
```

O operador **or**, por sua vez, apenas retornará **False** se ambas as afirmações forem falsas e **True** para qualquer outro caso.

```
Shell
>>> print(3 > 5 or 3 != 5)
True
>>> print(3 > 5 or 3 != 5)
True
>>> print(3 > 5 or 3 == 5)
False
>>> print(3 < 5 or 3 != 5)
True
```

Por último, o operador **not** serve basicamente para inverter (ou negar) o valor lógico de uma expressão. Isto é, expressões que retornariam **True** passarão a retornar **False**, e vice-versa, com o uso deste operador. Veja o exemplo abaixo:

```
Shell
>>> not True
False
>>> not False
True
>>> 5 == 5 and not 5 != 5
True
```


The background of the entire page is a dark blue field filled with a complex, light blue circuit-like pattern. This pattern consists of numerous thin, interconnected lines that branch out and terminate in small circular nodes, resembling a printed circuit board or a neural network diagram. The lines and nodes are more densely packed on the left side and become sparser towards the right.

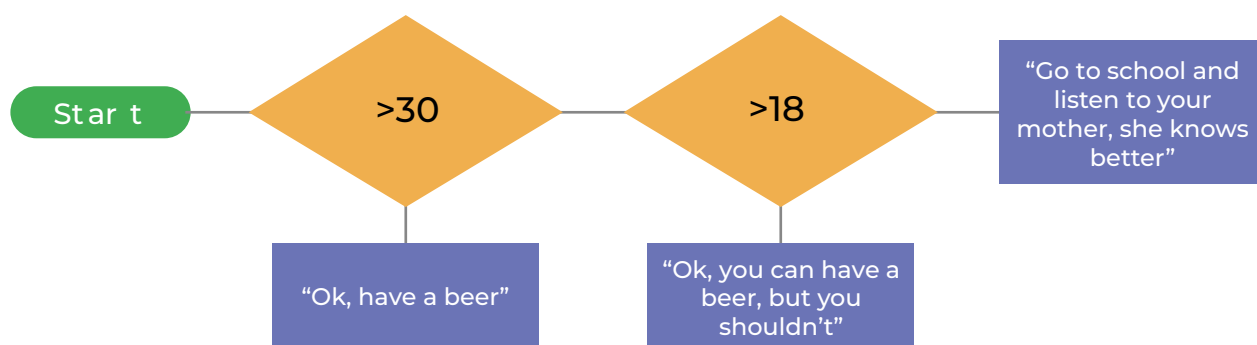
CAPÍTULO

03

CONTROLE
DE FLUXO

3.1 Condicionais

Em Python, cada linha de código é executada, uma por vez, em sequência, num fluxo até o fim. Os blocos condicionais que vamos estudar neste capítulo servem para redirecionar este fluxo de execução, de acordo com uma condição que é posta à prova.



O bloco de código que nos permite fazer esse tipo de teste em Python é o **if**. Após este bloco, basta descrever a condição a ser testada, seguida de dois pontos (:), e então listar o que deve ser feito caso o teste seja positivo:

```
example.py <
age = int(input("How old are you?"))
if age >= 18:
    print("Ok, you can drink, but you shouldn't")
if age < 18:
    print("Go to school and listen to your mother, she knows better")
```

Obs: Em Python, sempre que possuímos linhas de código “pertencentes” à definição de um bloco externo, utilizaremos o conceito de indentação para indicar essa relação. A indentação é a definição de um espaçamento padrão para cercar linhas de código relacionadas. É normal que usemos a tecla TAB para definir estes espaçamentos, esteja a um nível maior de tabulação do que o seu código “pai”.

Nesse exemplo, a terceira linha só será executada caso o usuário tenha idade maior ou igual a 18 anos, ao mesmo passo que a quinta linha só será processada se a idade for menor do que 18 anos. Repare que no exemplo, o segundo teste é exatamente o contrário do primeiro, enquanto um avalia se o usuário é maior de idade, o outro avalia se é menor. O Python fornece um aliado ao bloco **if** para nos ajudar em casos como esse, o **else**.

Usamos esse segundo bloco como uma forma de negar a condição do primeiro, isto é, caso a expressão proposta após o **if** retorne **False**, faremos o que é listado após o **else**. Observe o exemplo acima, reescrito com o uso desse novo bloco:

pois essa condição já foi negada a partir do **else** presente na linha 4.

Ainda podemos “enxugar” um pouco este código, utilizando um recurso que nos permite simplificar os casos em que precisamos usar mais um condicional **if** e logo após um bloco **else**. Este recurso é o **elif**. Reescrevendo o último exemplo, obtemos:

```
example.py
age = int(input("How old are you?"))
if age >= 30:
    print("Ok, have a beer")
elif age >= 18:
    print("Ok, you can drink, but you shouldn't")
else:
    print("Go to school and listen to your mother, she knows better")
```

3.2 While Loop

```
example.py
age = int(input("How old are you?"))
if age >= 18:
    print("Ok, you can drink, but you shouldn't")
else:
    print("Go to school and listen to your mother, she knows better")
```

Agora vamos aumentar um pouco o nosso código. Vamos adicionar uma terceira condição para usuários maiores que 30 anos de idade:

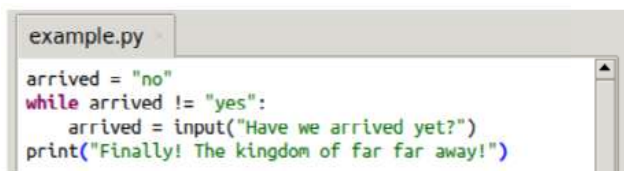
```
example.py
age = int(input("How old are you?"))
if age >= 30:
    print("Ok, have a beer")
else: # usuário tem idade menor ou igual a 29 anos
    if age >= 18:
        print("Ok, you can drink, but you shouldn't")
    else:
        print("Go to school and listen to your mother, she knows better")
```

Esse último caso é o que chamamos de condicional aninhado, quando temos mais de um conjunto de blocos **if-else**, dependentes entre si. Na linha 5 do exemplo acima, não precisamos testar se o usuário é maior de idade e tem idade inferior a 30 simultaneamente,

Ao longo de um programa, há algumas tarefas ou instruções que precisam ser executadas mais de uma vez. Para que isto não seja feito simplesmente com a repetição de linhas de código, o Python nos fornece diferentes tipos de **laços de repetição**, estruturas capazes de abrigar linhas de código que devem ser repetidas.

No caso em que a repetição de uma tarefa está condicionada ao valor de uma variável, à entrada do usuário no programa, ou algumas outras condições, temos um **While Loop**, também conhecido como laço condicional.

Esse bloco, portanto, deve ser precedido de um teste, tal qual o condicional **if**, e apenas com o resultado positivo, o código pertencente ao bloco de repetição será executado.



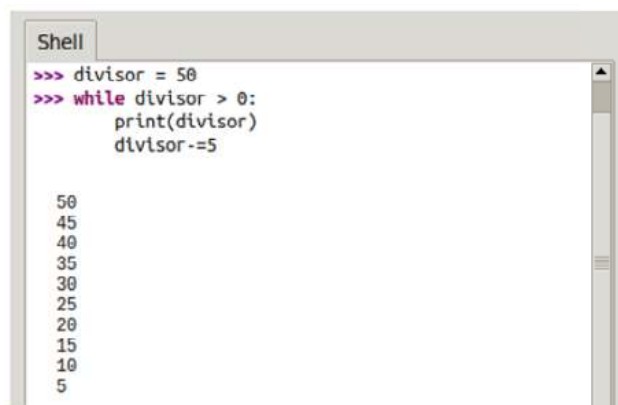
```
example.py
arrived = "no"
while arrived != "yes":
    arrived = input("Have we arrived yet?")
print("Finally! The kingdom of far far away!")
```

Repare que na primeira linha do nosso exemplo, nós damos um valor inicial à variável **arrived**, caso não o fizéssemos, teríamos um erro de variável não declarada. Este é um cuidado que você deve ter toda vez que usar o **while** com entrada de dados do usuário, como no caso acima. Uma das maneiras de contornar este problema seria chamar a função **input()** logo após o **while**, mas assim perderíamos a opção de usar o valor retornado mais tarde no programa.

No exemplo que utilizamos, caso o usuário confirmasse que havia chegado a “Tão Tão Distante” (no código escrito Far Far Away, em inglês) com qualquer expressão diferente de “yes”, o programa seria executado para sempre. Logo, tome sempre muito cuidado com o caminho que o seu programa pode trilhar quando estiver usando esse tipo de laço de repetição.

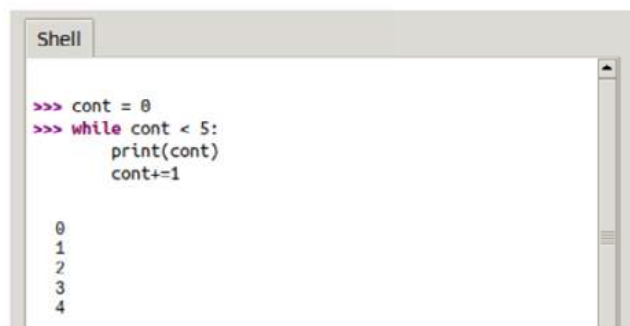
Um dos usos mais comuns do **While Loop** que você pode encontrar é o seu uso como contador, mais precisamente, como um repetidor finito a partir de um determinado número, com um limite definido. Nestes casos, usaremos sempre uma variável auxiliar que fará o papel de contador, sobre a qual realizaremos a condição que determina a parada do laço.

Esse uso do **while** é útil, não só quando precisamos que uma determinada sequência de comandos seja executada por uma quantidade de vezes pré definida, mas quando precisamos fazer uso desta variável contadora dentro do laço. Por exemplo, imagine que queiramos imprimir todos os divisores inteiros de 50, veja como poderíamos codificar para realizar esta ação em Python:



```
Shell
>>> divisor = 50
>>> while divisor > 0:
    print(divisor)
    divisor -= 5

50
45
40
35
30
25
20
15
10
5
```



```
Shell
>>> cont = 0
>>> while cont < 5:
    print(cont)
    cont += 1

0
1
2
3
4
```

The background of the entire page is a dark blue field filled with a complex, light blue circuit-like pattern. This pattern consists of numerous thin, interconnected lines that form a dense network, with small circular nodes at various points of intersection and termination. The overall effect is reminiscent of a printed circuit board or a digital data network.

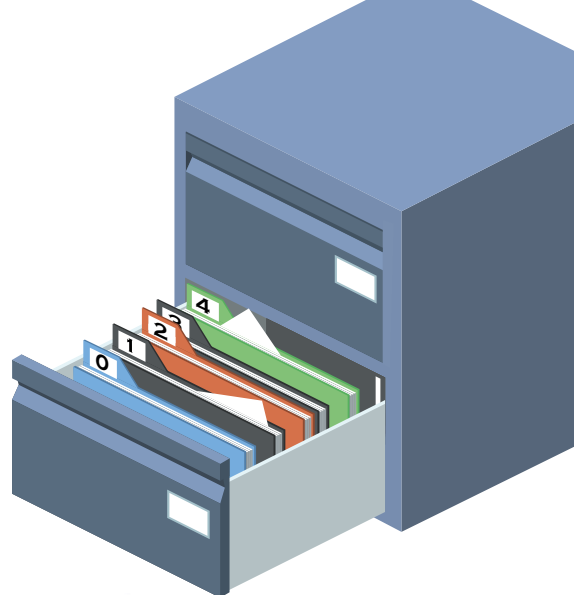
CAPÍTULO

04

ESTRUTURAS DE
DADOS: LISTAS

4.1 Introdução a Listas

Imagine que nós queiramos armazenar em uma variável o nome de todos os alunos de uma turma da Let's Code. Pelo que estudamos até agora, as variáveis só armazenam um valor por vez, portanto, uma das alternativas para solucionar este problema, seria utilizar o mesmo número de variáveis em relação a quantos alunos temos na turma, algo como:



```
Shell
>>> first_student, second_student, third_student, fourth_student = 'John', 'Mark', 'Laura', 'Marie'
```

É claro que fazer desta maneira seria extremamente ineficiente e exaustivo, afinal, imagine repetir este processo para uma turma de 20 alunos, ou então se o mesmo problema se apresentasse para armazenar o nome de todos os pagantes de ingressos para um jogo de futebol.

Para tratar essas situações, o Python fornece um tipo de dado em que é possível o armazenamento de um ou mais valores acessados pela mesma variável, esse tipo é a **lista**.

Para dar valores a uma lista, basta usarmos colchetes para delimitar a enumeração de valores que a variável deve conter, separando cada valor por uma vírgula, como no exemplo abaixo:

```
Shell
>>> students_list = ['John', 'Mark', 'Laura', 'Marie']
```

Você pode inclusive imprimir na tela toda uma lista e o retorno será muito similar à sua declaração. O mais comum, no entanto, é a impressão ou acesso a um determinado valor da lista. Também é possível referenciar um valor específico da lista através do seu **índice**, usando também a notação de colchetes. Os índices dos elementos em uma lista começam em 0 (e não em 1) e vão até a quantidade de itens - 1.

Essa convenção de indexação é chamada de indexação de base 0, ou simplesmente, indexação a partir de 0. Caso tenha dificuldade para associar esta ideia, basta pensar que o índice de cada item da lista, representa o quão distante este item está do início da lista. Tão logo, o primeiro elemento, está a 0 itens de distância do início, o segundo, a 1 item de distância, o terceiro a 2, e assim sucessivamente.

```
Shell
>>> print(students_list[0])
John
>>> # and if you want to print the last on the list
>>> print(students_list[3])
Marie
```

É possível ainda se utilizar uma indexação negativa, que permite o acesso de itens a partir do último elemento da lista. É importante perceber, no entanto, que apesar do primeiro item da lista ser referenciado pelo índice 0, o último elemento da lista será acessado a partir do índice -1 (e não -0), e os itens anteriores vão seguindo em ordem decrescente:

```
example.py
students_list = ['John', 'Mark', 'Laura', 'Marie']
print(students_list[-1])
print(students_list[-2])

Shell
>>> %Run example.py
Marie
Laura
```

Além de acessar índices individuais de uma lista, podemos também acessar uma subsequência da nossa lista (válido também para **strings**), utilizando a notação de fatiamento em Python:

```
example.py
week_days = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
'Saturday']
print(week_days[2:5])

Shell
>>> %Run example.py
['Tuesday', 'Wednesday', 'Thursday']
```

Repare que enquanto o primeiro índice é inclusivo, o último tem caráter excludente, ou seja, não é incluído na subsequência criada pelo fatiamento da lista. O fatiamento também funciona com índices negativos, inclusive combinando estes com índices positivos, como no exemplo abaixo:

```
example.py
week_days = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
'Saturday']

print(week_days[3:-1])

Shell

>>> %Run example.py
['Wednesday', 'Thursday', 'Friday']
```

Podemos ainda emitir o índice inicial ou final (ou ambos) no fatiamento de uma lista. Ao emitirmos o índice inicial, assume-se que este é 0, e ao emitirmos o índice final, o interpretador entende que queremos partir do índice inicial e ir até o fim da lista:

```
listas.py
week_days = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
'Saturday']

print(week_days[:2])
print(week_days[2:])

Shell

>>> %Run listas.py
['Sunday', 'Monday']
['Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']
```

4.2 Métodos de lista

A lista é uma estrutura de dados extremamente versátil, principalmente pelo seu caráter mutável que permite que a alteração de seus valores dinamicamente. Existem inúmeros métodos para trabalharmos com listas, os mais comuns e mais naturais desse tipo de dados são:

```
example.py
week_days = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
'Saturday']

week_days.append('Let's Code Day')
print("Appended Let's Code Day: ", week_days)

# Pode ser passado também um índice específico para ser removido
# se nada for passado, o método removerá o último item da lista
week_days.pop()
print("Removes the last element on the list: ", week_days)

week_days.pop(1)
print("Removes the element at index 1, cause nobody likes monday: ", week_days)

# Insere um elemento no índice especificado e afasta os itens para
# a direita desse índice
week_days.insert(1, 'Monday')
print("Inserts the element 'Monday' at the informed index (1): ", week_days)

week_days.remove('Sunday')
print("Removed the informed element: ", week_days)

Shell

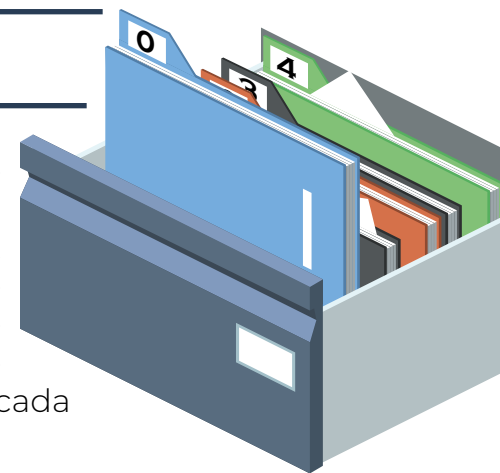
>>> %Run example.py
Appended Let's Code Day: ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Let's Code Day']
Removes the last element on the list: ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']
Removes the element at index 1, cause nobody likes monday: ['Sunday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']
Inserts the element 'Monday' at the informed index (1): ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']
Removed the informed element: ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']
```


No entanto, existem vários métodos para serem utilizados, segue abaixo uma tabela contendo alguns dos variados métodos e funções para trabalhar com listas:

| | |
|------------------------------------|---|
| <code>index(num)</code> | Retorna o índice do elemento passado por parâmetro ou dispara um erro caso não o encontre. |
| <code>clear()</code> | Apaga todos os elementos da lista, retornando a lista vazia. |
| <code>count(num)</code> | Conta o número de vezes em que o elemento passado por parâmetro é apresentado. |
| <code>reverse()</code> | Inverte a ordem dos elementos da lista. |
| <code>copy()</code> | Cria uma cópia da lista, retornando uma lista idêntica. |
| <code>sort()</code> | Ordena uma lista (seja com valores numéricos ou strings). |
| <code>random.shuffle(lista)</code> | Utilizando o módulo <code>random</code> , esta função embaralha a lista de modo aleatório. |
| <code>list(range(num))</code> | Cria uma lista de inteiros ordenados, a partir da função <code>range()</code> — melhor explicada no tópico For Loop, adiante no material. |
| <code>len(lista)</code> | Esta função retorna o número de elementos da lista passada por parâmetro. |

4.3 For Loop

É muito comum que tenhamos uma instrução que necessite se repetir por várias vezes sobre um determinado valor, como é o caso de operações com listas. Até agora, nós vimos alguns métodos padrão de lista para inserção e remoção de itens, mas isso não é nem de longe tudo que podemos fazer com esta estrutura de dados. A maior parte dessas operações é feita analisando ou trabalhando com cada item de uma lista.



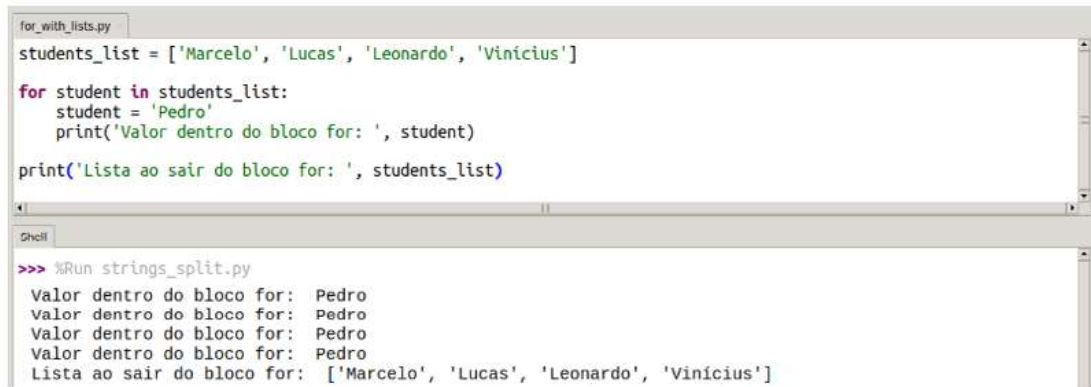
Nós já vimos como referenciar um item da lista diretamente através de seu índice. Mas numa lista grande, seria completamente inviável trabalhar com os índices um a um, da forma como conhecemos. Para resolver essa questão, usamos uma estrutura de repetição chamada **for**. Esse bloco trabalha com listas usando uma sintaxe muito simples.

```
Shell
>>> students_list = ['Marcelo', 'Lucas', 'Leonardo', 'Vinicius']
>>> for student in students_list:
    print(student)

Marcelo
Lucas
Leonardo
Vinicius
```

Logo após o **for**, nós damos um nome para como queremos referenciar o item da lista atual. No exemplo acima, cada item da lista de estudantes, nós chamamos de **student**.

É importante ressaltar, no entanto, que se fizermos qualquer alteração em um item da lista usando o **for**, da maneira como fizemos acima, estas alterações só valem para serem utilizadas dentro do bloco, conforme observamos abaixo:



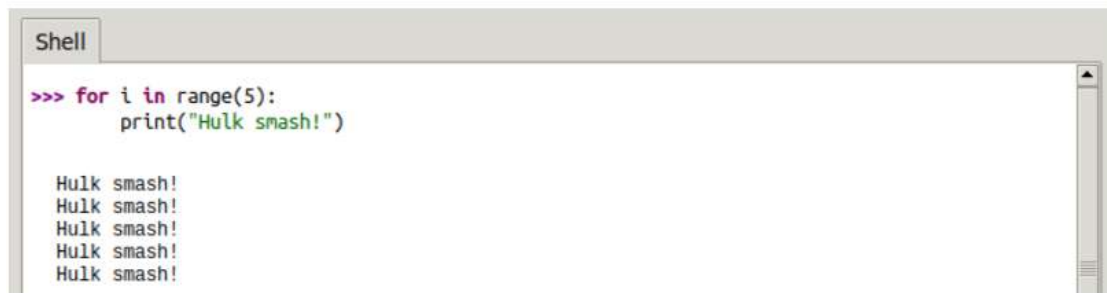
```
for_with_lists.py
students_list = ['Marcelo', 'Lucas', 'Leonardo', 'Vinicius']

for student in students_list:
    student = 'Pedro'
    print('Valor dentro do bloco for: ', student)

print('Lista ao sair do bloco for: ', students_list)

Shell
>>> %Run strings_split.py
Valor dentro do bloco for:  Pedro
Valor dentro do bloco for:  Pedro
Valor dentro do bloco for:  Pedro
Valor dentro do bloco for:  Pedro
Lista ao sair do bloco for:  ['Marcelo', 'Lucas', 'Leonardo', 'Vinicius']
```

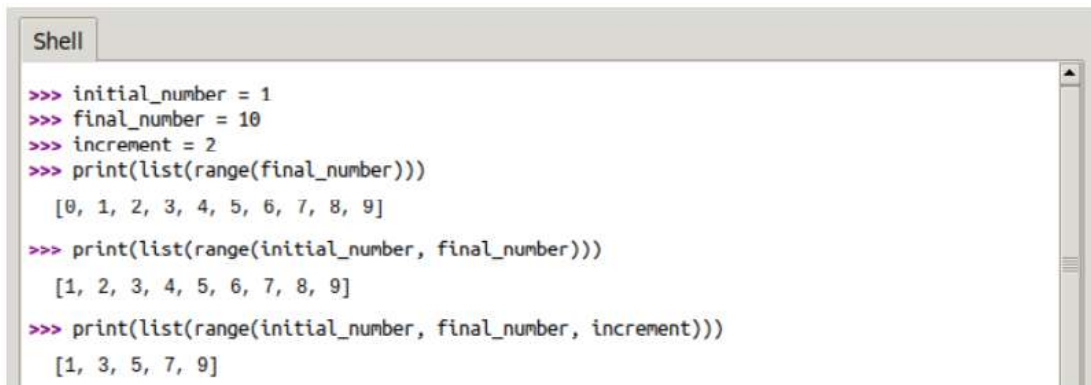
Outro uso comum dessa estrutura de repetição é o uso para repetir uma determinada tarefa por um número pré-definido de vezes, que pode ser realizado com a função **range()**. A função **range()** não retorna uma lista, mas um objeto iterável especial de tipo **range**.



```
Shell
>>> for i in range(5):
    print("Hulk smash!")

Hulk smash!
Hulk smash!
Hulk smash!
Hulk smash!
Hulk smash!
```

Você também pode usar a função **range()** para criar uma lista com uma sequência de números inteiros, passando seu retorno para a função **list()**, que cria a lista. Dependendo do número de parâmetros passados, a função **range()** pode ser utilizada de três formas diferentes, como mostrado abaixo:



```
Shell
>>> initial_number = 1
>>> final_number = 10
>>> increment = 2
>>> print(list(range(final_number)))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

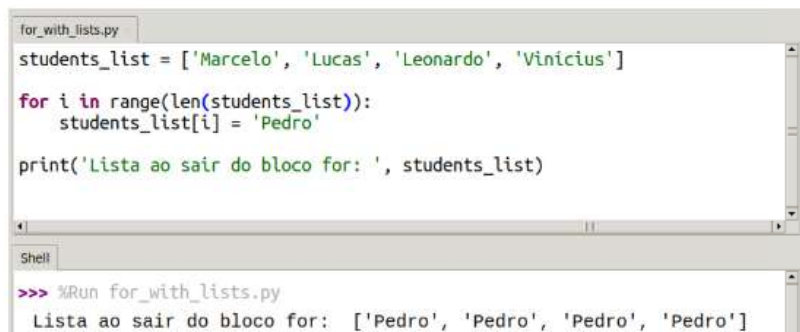
>>> print(list(range(initial_number, final_number)))
[1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> print(list(range(initial_number, final_number, increment)))
[1, 3, 5, 7, 9]
```

Quando passado apenas um parâmetro, cria-se uma lista de números inteiros de 0 ao número passado -1. Quando temos dois parâmetros, a lista criada vai do valor do primeiro parâmetro, até o segundo -1. Finalmente, com 3 parâmetros, o terceiro se torna um valor de incremento, então a lista é criada “pulando” os números, de incremento em incremento.

Obs: a função `range()` não aceita parâmetros de tipo **float**, apenas números inteiros.

Podemos usar a função **range()** para conseguir alterar os valores de elementos de uma lista dentro do **for**:



```
for_with_lists.py
students_list = ['Marcelo', 'Lucas', 'Leonardo', 'Vinicius']
for i in range(len(students_list)):
    students_list[i] = 'Pedro'
print('Lista ao sair do bloco for: ', students_list)

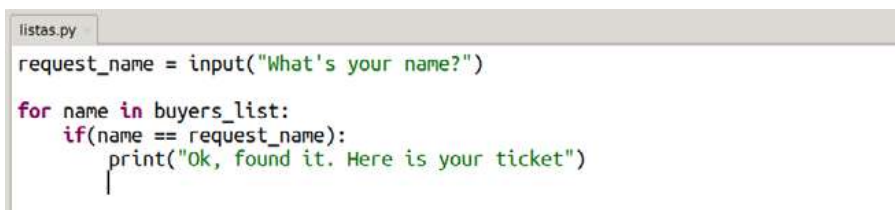
Shell
>>> %Run for_with_lists.py
Lista ao sair do bloco for: ['Pedro', 'Pedro', 'Pedro', 'Pedro']
```

Observe que dessa vez, o nosso **for** não itera sobre a lista de estudantes — que é a que queremos alterar —, mas sim sobre uma lista criada diretamente a partir do método **range()**, com elementos que vão de 0 até o tamanho da lista (retorno da função **len()**) - 1. Dentro do bloco, nós utilizamos esses valores para referenciar cada elemento com a notação de colchetes.

Caso você tenha ficado confuso com o exemplo, adicione aos comandos do **for** a linha **print(i)** para acompanhar como esses valores vão sofrendo alterações. Compare então com o resultado da iteração do mesmo código sem o uso do **for**, isto é, se alterássemos os elementos individualmente, linha a linha utilizando a mesma notação.

4.4 Comando break

Imagine o sistema de gerenciamento de um estádio de futebol. Agora pensemos que este sistema tenha como uma de suas funções a busca do nome dos torcedores, para checar a compra do ingresso prévia e se este já foi ou não retirado. Baseado no que vimos até agora, esse algoritmo seria algo como:



```
listas.py
request_name = input("What's your name?")
for name in buyers_list:
    if(name == request_name):
        print("Ok, found it. Here is your ticket")
```

No entanto, um jogo de futebol pode receber mais de 50.000 pessoas no estádio, e essa operação deve ser realizada para cada um dos torcedores, logo, são 50.000 repetições por torcedor (o número de repetições não se altera mesmo na hipótese do nome buscado ser o primeiro da lista). É claro que um sistema assim não seria eficiente, uma opção que pode ajudar na construção desse algoritmo é o comando **break**. Esse comando é usado para forçar a interrupção de um laço e é sempre associado com algum teste, como no nosso exemplo (ocorrendo justamente quando o nome buscado for encontrado pelo programa):

```
listas.py
request_name = input("What's your name?")

for name in buyers_list:
    if(name == request_name):
        print("Ok, found it. Here is your ticket")
        break
```

Deste modo, assim que o nome for encontrado, o laço é interrompido.

The background of the entire page is a dark blue field filled with a complex, light blue circuit-like pattern. This pattern consists of numerous thin, interconnected lines that branch out and terminate in small circular nodes, resembling a printed circuit board or a neural network diagram. The lines and nodes are more densely packed on the left side and become sparser towards the right.

CAPÍTULO

05

FUNÇÕES

5.1 Criando Funções

Desde a primeira linha de código escrita neste curso, estamos fazendo uso de funções, agora é a hora de criarmos as nossas próprias.

Funções são blocos de código que definimos para executar uma operação que utilizaremos mais de uma vez ao longo de nosso programa, a fim de facilitar a escrita do código responsável pela construção deste programa. Assim como nos últimos exemplos, houve a necessidade de inúmeras vezes utilizarmos as funções **print()**, para mostrar algo ao usuário; e **input()**, para captar uma entrada do mesmo.

```
example.py
def hello_world():
    print("Hello, world!")
```

A declaração de uma função em Python começa com a palavra reservada **def**, seguida do nome da nossa função (as regras para nomear uma função são as mesmas para nomeação de variáveis), e então o abrir e fechar de parênteses.

Para executar uma função que criamos em Python, basta escrever o seu nome, seguido dos parênteses:

```
example.py
def hello_world():
    print("Hello, world!")

hello_world()

Shell
>>> %Run example.py
Hello, world!
```

Os parênteses indicam passagem de argumentos (ou parâmetros) para nossa função. Os parâmetros de uma função são valores passados para que sejam utilizados dentro deste bloco de código de alguma maneira.

```
functions.py
def good_morning(name):
    print("Good morning,", name)

good_morning("Mr. Stark")

Shell
>>> %Run functions.py
Good mornind. Mr. Stark
```

A passagem de um parâmetro para uma função em Python é sempre feita por referência, o que significa que ao operarmos sobre um parâmetro passado à função, o seu valor fora dela também sofrerá alterações, como demonstrado abaixo:

```
example.py
def change_list(lst):
    lst[0] = 5

Shell
>>> lista = [8, 3, 4, 1]
>>> change_list(lista)
>>> print(lista)
[5, 3, 4, 1]
```

No entanto, quando dentro da função nós atribuímos um novo valor ao parâmetro, essa conexão entre o valor dentro e fora da função é quebrada e eles passam a ser valores completamente diferentes e desconexos entre si, veja:

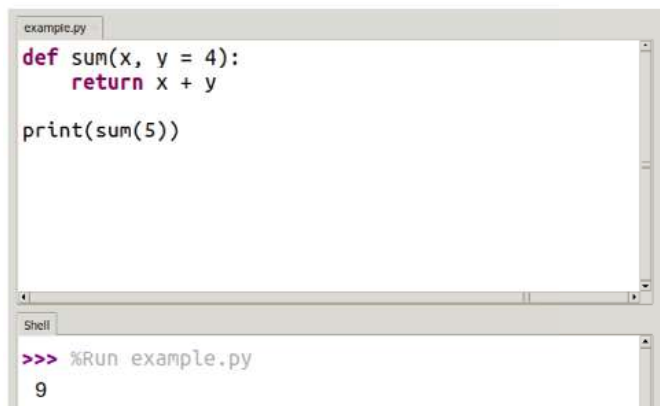
```
example.py
def change_list(lst):
    lst = [5, 6, 7]
    print("List inside the function:", lst)

lista = [8, 9, 10]
change_list(lista)
print("List outside the function:", lista)

Shell
>>> %Run example.py
List inside the function: [5, 6, 7]
List outside the function: [8, 9, 10]
```

5.2 Valores padrão (default)

Nem sempre precisamos passar todos os parâmetros a uma função, algumas delas podem ter um valor padrão a ser adotado caso o argumento não tenha sido passado. A sintaxe para estes casos é bem simples:



```
example.py
def sum(x, y = 4):
    return x + y

print(sum(5))

Shell
>>> %Run example.py
9
```

5.3 Valores padrão (default)

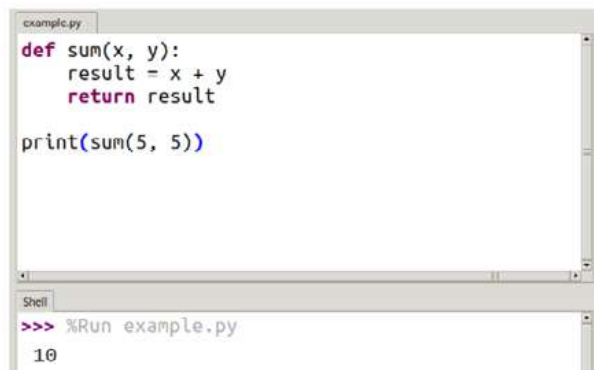
Quando chamamos uma função, é preciso passar a lista de argumentos exatamente na mesma posição em que foi previamente definida. Desta forma, **def xyz(nome, idade, altura)** precisa ser chamada, por exemplo, com uma sintaxe sequencial exata: **xyz("maria", 25, 1.65)**.

Porém, em Python, é possível alterar as posições, determinando, no ato da chamada, o nome de cada argumento. No nosso exemplo, a escrita do código ficará desta forma: **xyz(idade=25, altura=1.65, nome="maria")**.

5.4 Retorno de uma função

Falamos ainda há pouco sobre como as funções podem operar sobre valores passados, e naturalmente, como toda

operação obrigatoriamente tem um resultado, esse resultado pode ser retornado pela função. Por exemplo, tomemos uma função que retorna a soma de dois valores passados por parâmetro:



```
example.py
def sum(x, y):
    result = x + y
    return result

print(sum(5, 5))

Shell
>>> %Run example.py
10
```

Retornamos o valor de uma função através da palavra reservada **return**, embora tenhamos usado no exemplo acima uma variável auxiliar que guarda o resultado da operação, para que seja então retornado pela função. Em casos simples, como a soma de dois valores, podemos realizar a operação diretamente após o **return**:



```
example.py
def sum(x, y):
    return x + y
```

5.5 Recursividade

Um dos conceitos mais importantes em projeto de algoritmos é a recursividade. Um algoritmo recursivo é aquele que possui uma função que chama a ela mesma em algum momento de sua execução. Uma vez que isso ocorre, internamente se cria uma *pilha de execução*.

Toda vez que a função chamar a si mesma, a execução atual é abandonada e passa a esperar o

retorno de uma nova execução, que virá a partir desta nova chamada. É importante que lembremos que sempre deve haver um condicional que interrompa essas chamadas recursivas, para evitar que a execução se torne infinita.

```
functions.py
def factorial(number):
    factorial = 1
    for i in range(number, 1, -1):
        factorial *= i
    return factorial

def recursion_factorial(number):
    if number == 1:
        return
    return number * factorial(number - 1)

print(recursion_factorial(5))
print(factorial(5))

Shell
>>> %Run functions.py
120
120
```

Vamos usar como exemplo uma função que calcula o fatorial de um número, com e sem recursão:

Obs: A execução de um algoritmo recursivo **não pode** ser quebrada com um `break`, o comando `break` só deve ser usado dentro de um loop. A quebra de um algoritmo recursivo deve sempre ser feita com um retorno, embora esse retorno não precise ter um valor.

The background of the entire page is a dark blue color. It features a complex pattern of light blue lines that resemble circuit traces or data paths. These lines are interconnected by small circular dots, some of which are also light blue, while others are dark blue or black. The overall effect is a technical, digital aesthetic.

CAPÍTULO

06

TIPOS DE DADOS:
STRINGS

6.1 Mais poderes com Strings

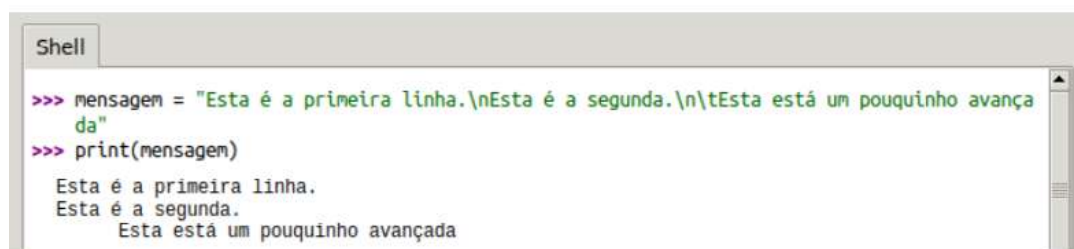
Falamos sobre como podemos delimitar um valor textual, usando aspas simples ou duplas, e estas duas opções cobrem boa parte das construções textuais mais simples, ao criar um texto que possua uma citação, podemos delimitar a **string** com aspas simples e usar aspas duplas para a citação, já quando precisarmos do apóstrofo, delimitar o texto com aspas duplas. Porém, com a evolução da complexidade das construções do texto do código, haverá momentos em que será necessário utilizar o caractere delimitador como conteúdo do texto.

Para contornar esse problema, podemos usar o caractere de “escape” (\) evitando o conflito entre as aspas que devem delimitar o valor da string e as que devem ser parte do conteúdo. A escrita pode ocorrer da seguinte maneira:



```
Shell
>>> print("\"I am Groot\"", said Groot)
"I am Groot", said Groot
```

Além das aspas, o escape também permite a inclusão de indicadores especiais de texto, como a quebra de linha ou a tabulação, por exemplo.

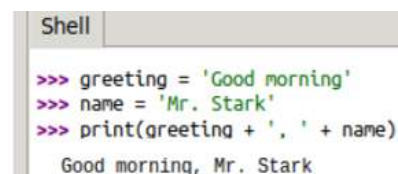


```
Shell
>>> mensagem = "Esta é a primeira linha.\nEsta é a segunda.\n\tEsta está um pouquinho avançada"
>>> print(mensagem)
Esta é a primeira linha.
Esta é a segunda.
    Esta está um pouquinho avançada
```

No exemplo acima, podemos observar que a quebra de linha é o **\n** e o TAB é o **\t**. Quando o interpretador encontra a barra invertida, ele interpreta o que vem imediatamente após como uma instrução a ser seguida. Portanto, é preciso conhecer bem o que pode ser usado depois da barra:

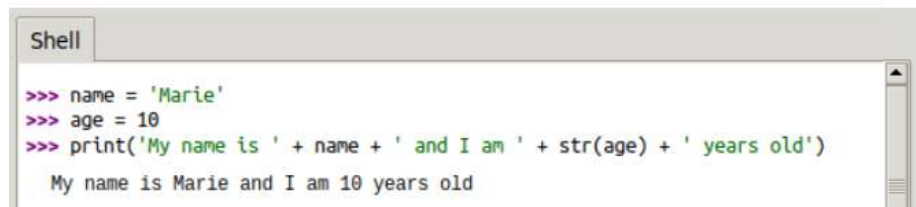
| | |
|-----------------|--|
| <code>\t</code> | Inserir um tab no texto nesse lugar |
| <code>\b</code> | Inserir um espaço no texto |
| <code>\n</code> | Inserir uma nova linha no texto |
| <code>\r</code> | Inserir um retorno de cartucho no texto |
| <code>\'</code> | Inserir uma aspa simples (') no texto |
| <code>\"</code> | Inserir uma aspa dupla (") no texto |
| <code>\\</code> | Inserir uma barra invertida (\) no texto |

O operador de adição que utilizamos no tópico de **Integers** e **Floats** pode também ser usado para **strings** para concatenar diferentes valores textuais. Como por exemplo:



```
Shell
>>> greeting = 'Good morning'
>>> name = 'Mr. Stark'
>>> print(greeting + ', ' + name)
Good morning, Mr. Stark
```

No entanto, se tentarmos concatenar uma **string** com o valor numérico, o interpretador irá apontar um erro de conflito de tipos. Para resolver esta situação, podemos converter este valor numérico em **string**, utilizando a função de **casting** **str()**. Veja como, abaixo:



```
Shell
>>> name = 'Marie'
>>> age = 10
>>> print('My name is ' + name + ' and I am ' + str(age) + ' years old')
My name is Marie and I am 10 years old
```

Pode-se também utilizar operador de multiplicação para **strings**, tal operador permite que aquele texto se repita pelo número de vezes que for multiplicado, da seguinte maneira:



```
Shell
>>> groot_only_saying_ever = 'I am Groot'
>>> print(groot_only_saying_ever * 10)
I am GrootI am GrootI am GrootI am GrootI am GrootI am GrootI am GrootI am Gro
otI am Groot
```

Já os operadores de subtração e divisão, por sua vez, vão gerar um erro de conflito entre tipos, se forem utilizados junto das strings [negrito]. Faça o teste para observar como este erro ocorre!

6.2 Strings como listas

Em vários aspectos, as **strings** se comportam como listas, como pudemos observar quando falamos sobre o fatiamento de dessa estrutura de dados, por exemplo. Porém, o fatiamento não é a única operação de listas que funciona também para as **strings**. O acesso a índices individuais também é possível em strings, cada caractere sendo acessado por um índice diferente:



```
strings.py
language = "Python"
print(language[0])
print(language[-1])
print(language[2:4])

Shell
>>> %cd '/var/www/python/apostila python'
>>> %Run strings.py
P
n
th
```

A lógica aqui é similar ao uso do **For Loop** em listas, em que a cada iteração acessamos um elemento, o mesmo acontece com **strings**, acessando um caractere por vez:

```
strings_for.py
language = "Python"

for character in language:
    print(character)

Shell
>>> %Run strings_for.py
P
y
t
h
o
n
```

É importante, no entanto, ressaltar que ao contrário das listas, as **strings** não permitem que seus elementos (caracteres) sejam alterados diretamente:

```
Shell
>>> language[3] = "x"
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>>
```

6.3 Métodos de Strings

Métodos diferem de funções pois são aplicados a partir da variável, alterando diretamente o seu valor, enquanto as funções recebem um valor por parâmetro e retornam o resultado de operação que realizam, em um processo diferente por natureza. Alguns exemplos mais comuns de métodos para **strings** são:

| | |
|---------------------------|--|
| <code>upper()</code> | converte todos os caracteres para caixa alta |
| <code>lower()</code> | converte todos os caracteres para letras minúsculas |
| <code>capitalize()</code> | converte para caixa alta apenas a primeira letra do texto |
| <code>title()</code> | converte para caixa alta a primeira letra de cada palavra do texto |

Já falamos sobre utilizar **strings** como listas, mas há uma outra abordagem que pode nos dar um poder muito grande de operações. Se trata do uso de **strings** como os dados de uma lista. Um uso comum deste modelo é a separação de palavra por palavra de uma **string**, armazenando cada palavra como um elemento separado da lista. Uma forma simples de realizar esta ação é através do método **split()**, conforme demonstrado abaixo:

```
strings_split.py
months = 'January February March April May June July August September October November December'.split()
print(type(months))
for month in months:
    print(month)

Shell

>>> %Run strings_split.py
<class 'list'>
January
February
March
April
May
June
July
August
September
October
November
December
```

Quando não passamos nenhum parâmetro ao método `split()`, ele separa palavra por palavra, considerando o espaço “ ” como caractere delimitador de cada elemento. No entanto, se quiser utilizar um caractere mais específico para separar seus elementos, basta passá-lo como parâmetro para o método:

```
strings_split.py
months = 'January:February:March:April:May:June:July:August:September:October:November:December'.split(':')
print(type(months))
for month in months:
    print(month)

Shell

>>> %Run strings_split.py
<class 'list'>
January
February
March
April
May
June
July
August
September
October
November
December
```

Outro método que opera sobre **strings** e listas é o **join()**. Trata-se de um método de **string** que adiciona seu valor a cada elemento da lista (exceto o último) passada como parâmetro:

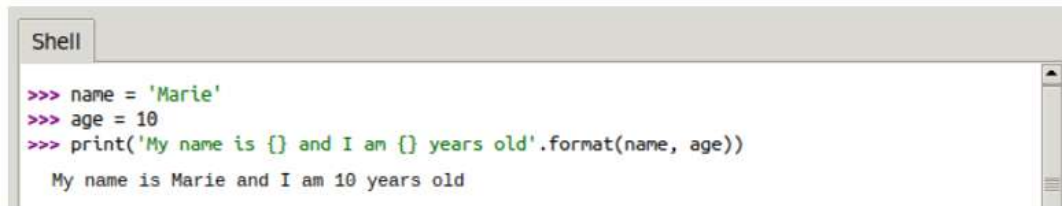
```
strings_join.py
week_days = 'Monday Tuesday Wednesday Thursday Friday Saturday'.split()
suffix = ' is a work day, '
final_message = suffix.join(week_days)
print(final_message)

Shell

>>> %Run strings_join.py
Monday is a work day, Tuesday is a work day, Wednesday is a work day, Thursday is a work day, Friday
is a work day, Saturday
```

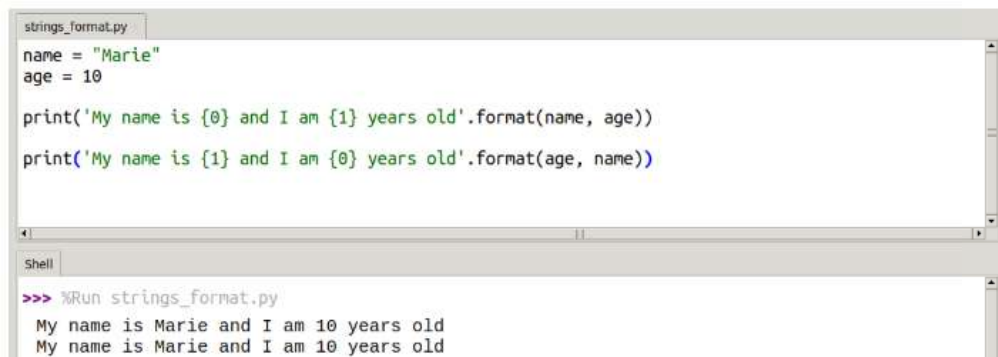
Uma das operações mais comuns em strings é a formatação das mesmas, que pode ser feita por uma grande diversidade de métodos. Mas de maneira geral, esse conceito se traduz na ideia de interpolar valores de variáveis dentro do texto.

O método para formatação de strings mais usado em Python é o `.format()`, que facilita a construção de strings, permitindo posicionar o valor de uma variável direto no texto. Com este método, todas as vezes em que uma variável é citada, podemos substituir pelo abrir e fechar de chaves, informando em seguida ao método, qual é a variável que estamos nos referindo.



```
Shell
>>> name = 'Marie'
>>> age = 10
>>> print('My name is {} and I am {} years old'.format(name, age))
My name is Marie and I am 10 years old
```

Bem mais simples, não?! Essas chaves vão seguir a mesma ordem a que são passados os valores para o método **.format()**. É possível, no entanto, fazer referência aos parâmetros do método dentro da **string**, através de índices informados entre as chaves. Os índices seguem a mesma regra que estudamos em listas, isto é, indexação a partir do zero, veja abaixo.

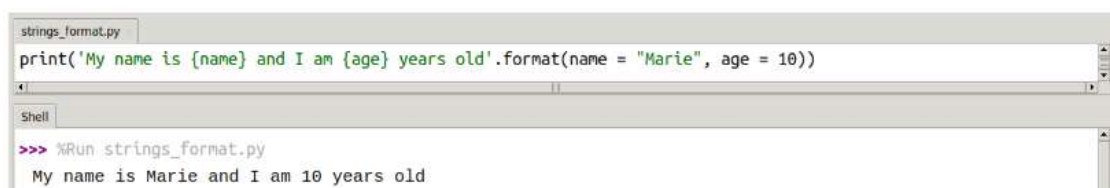


```
strings_format.py
name = "Marie"
age = 10

print('My name is {0} and I am {1} years old'.format(name, age))
print('My name is {1} and I am {0} years old'.format(age, name))

Shell
>>> %Run strings_format.py
My name is Marie and I am 10 years old
My name is Marie and I am 10 years old
```

Outra maneira de utilizar o método **format** é usando índices nomeados, parecido com o que fizemos em funções, quando fazíamos a chamada informando uma sequência de pares chave/valor:



```
strings_format.py
print('My name is {name} and I am {age} years old'.format(name = "Marie", age = 10))

Shell
>>> %Run strings_format.py
My name is Marie and I am 10 years old
```

O método **format** possui uma declaração especial, um shorthand extremamente fácil de utilizar, que não precisa que se passe os parâmetros. Este atalho para o método é chamado de **f-strings** e é também uma forma de interpolar diretamente o valor da variável na **string**, com o bônus de poder utilizar diretamente o nome da variável, sem o uso de parâmetros chave/valor, da seguinte maneira:

strings_format.py

```
name = 'Marie'  
age = 10  
  
print(f'My name is {name} and I am {age} years old')
```

Shell

```
>>> %Run strings_format.py  
My name is Marie and I am 10 years old
```

The background of the entire page is a dark blue field filled with a complex, light blue circuit-like pattern. This pattern consists of numerous thin, interconnected lines that form a dense web of paths, with small circular nodes at various points of intersection and termination. The lines vary in thickness and orientation, creating a sense of depth and technical complexity.

CAPÍTULO

07

ARQUIVOS

7.1 Arquivos como já conhecemos

Para que um programa seja realmente útil, ele precisa trabalhar com dados do “mundo real”, como imagens, vídeos, bases de dados, páginas de websites, toda essa informação é armazenada em arquivos. Os dados com que trabalhamos até agora, foram declarados direta-

mente no código do nosso programa ou então informados pelo usuário em tempo de execução.

Todos os diferentes tipos de arquivos com os quais estamos acostumados são na verdade bem parecidos. Seja um arquivo de imagem, ou uma página web, o arquivo responsável por ambos nada mais é do que uma sequência de caracteres com uma formatação própria indicativa do dado que representa seguido de uma extensão que identifica o tipo do arquivo, seja png, jpeg, html, mp3, mp4 ou qualquer outro que você conheça.

Estamos acostumados, portanto, a mover, criar, e manipular esses arquivos na nossa rotina digital. Neste capítulo, começaremos a expandir esse poder manipulando arquivos através do Python.

7.2 Manuseando arquivos com Python

Provavelmente o tipo de arquivo mais simples com que você já teve contato foi o formato padrão da maioria dos editores de textos mais simples, o txt. Este tipo de arquivo tem formatação fácil, sendo basicamente o que chamamos de plain text, traduzido como “apenas texto”.

A primeira função que nós precisamos estudar em Python para começar a manipulação de arquivos é a responsável pela abertura (em alguns casos, criação) de um arquivo, que é a função **open()**.

Vamos criar um arquivo .txt na mesma pasta do nosso arquivo .py com um texto simples, “Hello, world”, para não perder o costume:



Texto sem formatação ▾ Largura da tabulação: 8 ▾ Lin 1, Col 13 ▾ INS

Para acessar este arquivo e ler o seu conteúdo usando Python, basta utilizarmos as seguintes linhas de código:

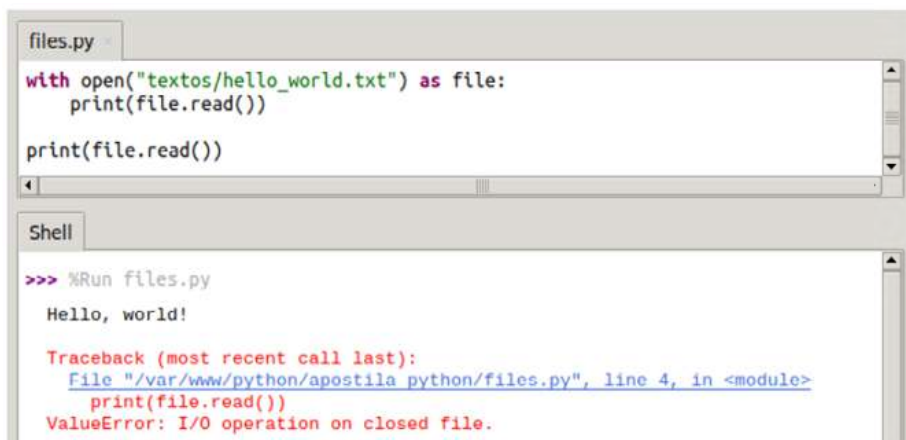


A função **open()** precisa receber ao menos um parâmetro, sendo este o nome do arquivo que se quer acessar, para retornar uma referência do arquivo aberto para que possamos ler ou escrever sobre o mesmo.

Obs: Há duas maneiras de passar o arquivo para a função **open()**, uma é usando o caminho relativo, isto é, o caminho do arquivo a partir da pasta do arquivo Python (.py), a outra é o que chamamos de caminho absoluto, que é o caminho do arquivo a partir da pasta raiz do sistema. Se quiser utilizar o caminho absoluto, comece a indicação por uma barra (/) e então complete o caminho.

O método **read()** retorna o conteúdo completo do arquivo em formato de **string**. Finalmente, temos mais um método que não pode nunca ser esquecido, o método **close()**. Essa lembrança de sempre fechar o arquivo que estamos trabalhando no Python é muito importante, pois é assim que destruímos a referência que foi criada para a manipulação dele. Podemos até não fechar os arquivos no nosso programa, e o código ainda funcionará perfeitamente, mas isso pode ter consequências, especialmente na segurança do nosso programa.

Uma maneira de contornar a necessidade de policiar o fechamento de um arquivo toda vez que precisarmos operar sobre ele, é utilizando o bloco **with**. Usando este bloco, nós podemos operar sobre um objeto do programa com sua referência apenas disponível para os comandos dentro do bloco. Em outras palavras, não precisaríamos mais usar o método **close()** para destruir esta referência.



```
files.py
with open("textos/hello_world.txt") as file:
    print(file.read())

print(file.read())

Shell
>>> %Run files.py
Hello, world!

Traceback (most recent call last):
  File "/var/www/python/apostila_python/files.py", line 4, in <module>
    print(file.read())
ValueError: I/O operation on closed file.
```

Repare que o **print** dentro do bloco **with** funciona perfeitamente, mas o que está fora do bloco, retorna um erro de leitura de tentar fazer uma operação em um arquivo fechado, indicando o fechamento automático do arquivo após o bloco **with**.

O modo padrão de abertura de um arquivo é o de **somente leitura**, ou read only. Para escrevermos sobre um arquivo, precisamos especificar para a função **open()** o modo de abertura que desejamos, no caso de escrita, o modo **'w'**, ou **write**.



```
files.py
file = open("hello_world.txt", "w")
file.write("I am Ultron")
file.close()
```

Para testar, execute as linhas de código acima (você pode mudar o texto a ser escrito, claro), e abra de novo o seu arquivo para ver a mudança. O modo **“w”** de abertura seguido do método **write()**, substitui o conteúdo do arquivo aberto pelo que acaba de ser informado pelo método.

Se o que você está buscando é apenas adicionar texto a um arquivo. O modo de abertura que você procura é o modo **append**, ou **“a”**.

```
files.py
file = open("hello_world.txt", 'a')
file.write(" and I'm gonna kill y'all!")
file.close()
```

Utilize o modo **append**, faça o teste e abra novamente o seu arquivo. Agora o texto informado ao método **write()** será adicionado ao fim do arquivo ao invés de substituir o existente.

A tabela completa de modos de abertura segue abaixo:

| | |
|-----|---|
| 'r' | abre o arquivo somente para leitura |
| 'w' | abre o arquivo para escrita, exclui o conteúdo do arquivo |
| 'x' | open for exclusive creation, failing if the file already exists |
| 'a' | open for writing, appending to the end of the file if it exists |

Você pode combinar os modos de abertura usando o operador **“+”**, por exemplo, para abrir o arquivo para leitura em modo binário, informe o modo de abertura como **“r+b”**.

É importante dizer que as formatações que vimos em **strings**, para quebra de linha, tabulação, retorno, entre outras, se aplicam aqui também.

Um uso interessante dos arquivos em Python é a possibilidade de utilizar o **For Loop** para acessar linha a linha, desta maneira:

```
files.py
"""
arquivo names.txt

Leonardo
Donatello
Rafael
Michelangelo
"""
file = open("textos/names.txt", 'r')

for ninja_turtle in file:
    print(ninja_turtle)

file.close()
```

```
Shell
>>> %Run files.py
Leonardo
Donatello
Rafael
Michelangelo
```

Outra forma de obter um resultado parecido é utilizando o método **readline()** que como o próprio nome sugere, lê uma linha por vez do nosso arquivo:

```
files.py
"""
arquivo names.txt

Leonardo
Donatello
Rafael
Michelangelo
"""
file = open("textos/names.txt", 'r')

for i in range(4):
    print(file.readline())

file.close()
```

```
Shell
>>> %Run files.py
Leonardo
Donatello
Rafael
Michelangelo
```

Repare que desta vez não fizemos a iteração do **for** sobre o arquivo. Se fizéssemos isso, o próprio **for** trataria de vagar pelas linhas do nosso arquivo

em conjunto com o método **readline()**, o que acabaria fazendo com que o algoritmo pulasse algumas linhas. Por isso utilizamos o **for** apenas para executar o **readline()** 4 vezes no exemplo acima.

Quando estamos realizando operações de leitura sobre um arquivo, o algoritmo usa uma espécie de “cursor” que se desloca pelo arquivo a medida que o lemos, logo, cada vez que o **readline()** é executado, este cursor é transportado para o fim da linha que acabou de ser lida. O mesmo vale para o método **read()**, mas nesse caso o cursor vai diretamente para o final já que o método lê o arquivo inteiro quando chamado.

Abaixo seguem alguns métodos de escrita, leitura, e controle do cursor:

| | |
|---|--|
| <code>file.read()</code> | Lê todo o conteúdo do arquivo |
| <code>file.readline()</code> | Lê e retorna a próxima linha do texto |
| <code>file.readlines()</code> | Lê e retorna todas as linhas do texto no formato de lista |
| <code>file.write(texto)</code> | Escreve o texto no arquivo |
| <code>file.writelines(lista_de_linhas)</code> | Escreve uma lista de strings em um arquivo de forma sequencial |
| <code>file.tell()</code> | Retorna o número do caractere que você está lendo no arquivo |
| <code>file.seek(pos)</code> | Muda o cursor do arquivo para uma posição |

7.3 Arquivos CSV

Talvez você já tenha percebido que alguns softwares de criação e edição de planilhas (Excel, Google Spreadsheets) ao te dar a opção de exportar ou salvar um arquivo, oferecem a opção pelo formato csv (comma separated values – valores separados por vírgula). Os arquivos csv são usados para armazenamento de dados de forma muito similar ao que normalmente usamos em planilhas:

The screenshot shows a Google Sheet titled "Censo 2019" with the following data:

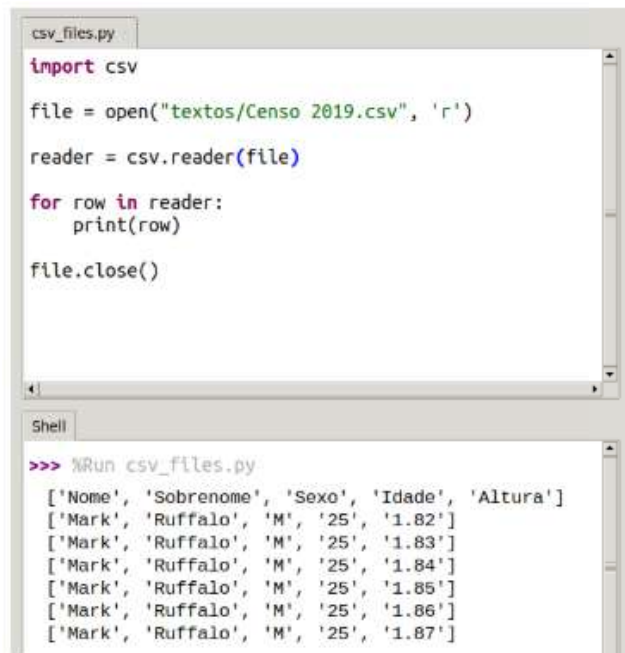
| | A | B | C | D | E |
|---|------|-----------|------|-------|--------|
| | Nome | Sobrenome | Sexo | Idade | Altura |
| 1 | Mark | Ruffalo | M | 25 | 1.82 |
| 2 | Mark | Ruffalo | M | | 1.83 |
| 3 | Mark | Ruffalo | M | | 1.84 |
| 4 | Mark | Ruffalo | M | | 1.85 |
| 5 | Mark | Ruffalo | M | | 1.86 |
| 6 | Mark | Ruffalo | M | | 1.87 |

Overlaid on the sheet is a CSV export window titled "Censo 2019 - Página1.csv" showing the data in CSV format:

```
Nome,Sobrenome,Sexo,Idade,Altura
Mark,Ruffalo,M,25,1.82
Mark,Ruffalo,M,25,1.83
Mark,Ruffalo,M,25,1.84
Mark,Ruffalo,M,25,1.85
Mark,Ruffalo,M,25,1.86
Mark,Ruffalo,M,25,1.87
```

Arquivos csv podem ser grandes aliados para armazenamento de dados - a própria similaridade com os arquivos editados em formato de planilha nos mostra isso - e como você pode ver, sua formatação é muito simples e fácil de trabalhar sem nem mesmo precisar de um editor de planilhas, por exemplo.

Em Python, há uma biblioteca (também chamada de módulo) específica para trabalhar com este tipo de arquivo. Para usá-la, basta usar o comando padrão de importação de módulos, o **import**.



```
csv_files.py
import csv

file = open("textos/Censo 2019.csv", 'r')

reader = csv.reader(file)

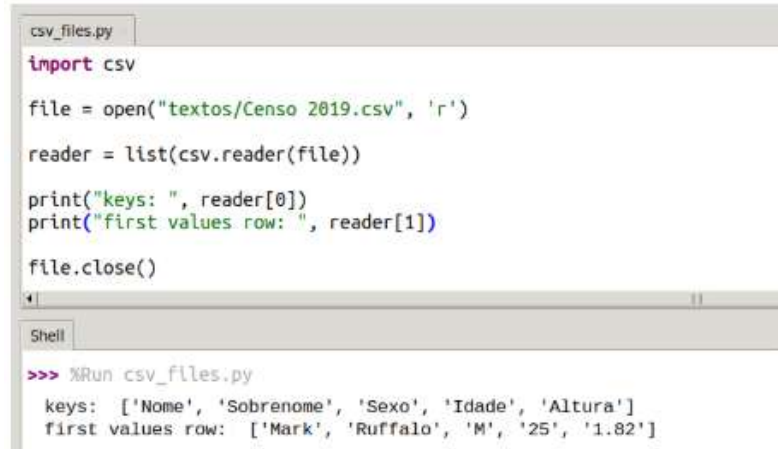
for row in reader:
    print(row)

file.close()
```

```
Shell
>>> %Run csv_files.py
['Nome', 'Sobrenome', 'Sexo', 'Idade', 'Altura']
['Mark', 'Ruffalo', 'M', '25', '1.82']
['Mark', 'Ruffalo', 'M', '25', '1.83']
['Mark', 'Ruffalo', 'M', '25', '1.84']
['Mark', 'Ruffalo', 'M', '25', '1.85']
['Mark', 'Ruffalo', 'M', '25', '1.86']
['Mark', 'Ruffalo', 'M', '25', '1.87']
```

Vamos começar com a leitura de um arquivo csv. Embora você possa usar todas as funções e métodos que já vimos em arquivos para manipular um csv, essa biblioteca padrão já oferece algumas funcionalidades específicas para as nossas operações finais, quando se trata deste tipo de arquivo. Por exemplo, o método `reader()` da biblioteca nos devolve as linhas do arquivo já em formato de listas, o que facilita a nossa operação sobre esses valores.

Algo muito comum de se utilizar com o uso do **reader()** é o casting do retorno desse método para uma lista, e então, teremos em mãos já uma lista para acesso direto às linhas do arquivo, sem precisar do **for**, veja abaixo:



```
csv_files.py
import csv

file = open("textos/Censo 2019.csv", 'r')

reader = list(csv.reader(file))

print("keys: ", reader[0])
print("first values row: ", reader[1])

file.close()
```

```
Shell
>>> %Run csv_files.py
keys: ['Nome', 'Sobrenome', 'Sexo', 'Idade', 'Altura']
first values row: ['Mark', 'Ruffalo', 'M', '25', '1.82']
```

Embora a formatação padrão do csv seja a separação dos valores por vírgula, alguns softwares exportam para este formato usando outros caracteres como delimitadores, o Excel, por exemplo, separa os valores usando ";". Você pode facilmente informar isso ao método **reader()**, basta adicionar o parâmetro `delimiter`. Por exemplo:

```
csv.reader(file, delimiter = ';')
```

Para escrever arquivos csv, também utilizaremos a biblioteca padrão do Python. Existem basicamente dois métodos de escrita para arquivos csv a partir dessa biblioteca, o **writerow()** – para escrever apenas uma linha, e o **writerows()** – para a escrita de mais de uma linha por vez. Ambos recebem como parâmetro listas que serão escritas no arquivo, o **writerow()** recebe uma lista simples, e o **writerows()** uma lista de listas:

```

csv_files.py
import csv

file = open("textos/Censo 2019.csv", 'w')


writer = csv.writer(file)

keys = ['Nome', 'Peso', 'Sexo', 'Idade']
writer.writerow(keys)

rows = [
    ['Leo', 82, 'M', 26],
    ['Pietro', 65, 'M', 20],
    ['Marcelo', 66, 'M', 24]
]
writer.writerows(rows)

file.close()

```

Abzir ▾  **Censo 2019.csv**
 /var/www/python/apostila pyth

| Nome | Peso | Sexo | Idade |
|---------|------|------|-------|
| Leo | 82 | M | 26 |
| Pietro | 65 | M | 20 |
| Marcelo | 66 | M | 24 |

Mais uma vez, podemos também alterar o delimitador padrão dos arquivos passando o padrão **delimiter**, dessa vez para o método **writer**.

The background of the entire page is a dark blue color with a complex, light blue circuit board pattern. The pattern consists of numerous thin, interconnected lines and small circular nodes, resembling a printed circuit board (PCB) layout. The lines and nodes are more densely packed on the left side and become sparser towards the right.

CAPÍTULO

08

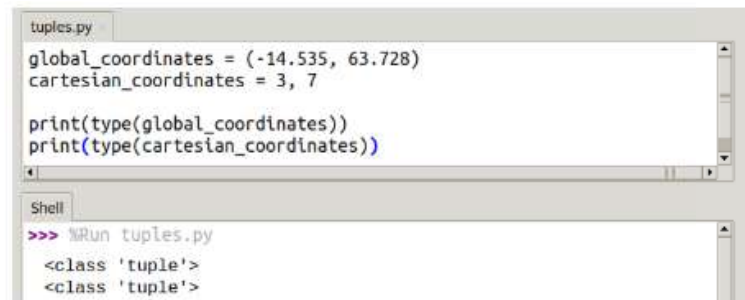
ESTRUTURAS DE
DADOS II: TUPLAS E
DICIONÁRIOS

8.1 Tuplas (tuples)

Existe um tipo de lista especial, chamado tupla, que é muito parecida com uma lista convencional, porém seu conteúdo, uma vez definido, é imutável. Esta estrutura de dados é usada para definir valores que são relacionados entre si, mas não são a mesma coisa. Nos nossos exemplos de lista, sempre definimos um conjunto de valores do mesmo tipo, como "meses", "dias da semana", ou seja, dados que continham a mesma informação, além disso, estes dados eram de fácil manipulação, podíamos adicionar, remover, alterar cada um deles sem problemas.

No caso da tupla, os elementos e suas respectivas posições, uma vez definidas, não variam mais. Aqui pode surgir a pergunta: "por quê então usar tuplas se são como listas, mas com menos recursos?", as tuplas são normalmente utilizadas para armazenar valores com alta proximidade conceitual, mas que não são a mesma coisa. Como por exemplo, coordenadas x e y, de latitude e longitude, valores RGB de uma cor, entre outros exemplos. Além disso, as tuplas ocupam menos memória que uma lista, então a escolha entre estas diferentes estruturas é uma questão não só de semântica, mas também de otimização de uso de memória por nosso programa.

A declaração de uma tupla pode ser feita usando parênteses, ou apenas listando os diferentes valores que a compõe, da seguinte maneira:



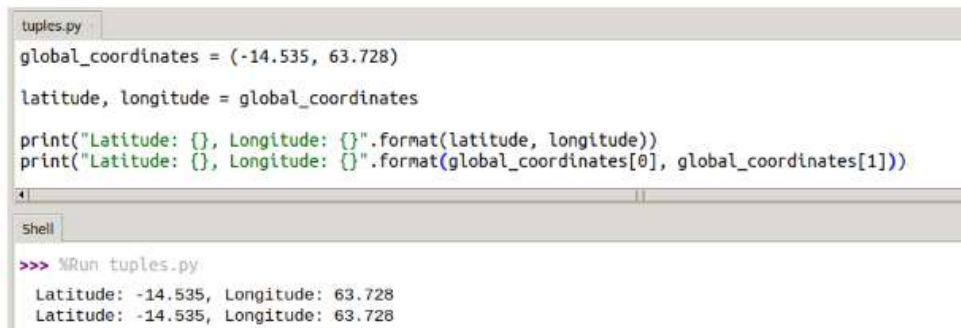
```
tuples.py
global_coordinates = (-14.535, 63.728)
cartesian_coordinates = 3, 7

print(type(global_coordinates))
print(type(cartesian_coordinates))

Shell
>>> %Run tuples.py
<class 'tuple'>
<class 'tuple'>
```

O acesso aos elementos de uma tupla, é feito da mesma maneira que em listas, basta usar a notação de colchetes e indicar o índice a que se refere o elemento.

As tuplas podem ser usadas para atribuir valores a diferentes variáveis de uma vez só (processo chamado de desempacotamento), assim como vimos no tópico de variáveis:



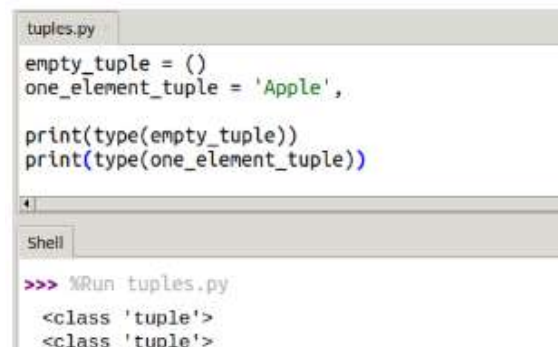
```
tuples.py
global_coordinates = (-14.535, 63.728)
latitude, longitude = global_coordinates

print("Latitude: {}, Longitude: {}".format(latitude, longitude))
print("Latitude: {}, Longitude: {}".format(global_coordinates[0], global_coordinates[1]))

Shell
>>> %Run tuples.py
Latitude: -14.535, Longitude: 63.728
Latitude: -14.535, Longitude: 63.728
```

Assim como as listas, uma tupla pode conter valores de diferentes tipos, inclusive, listas. Alterar o valor de um elemento de uma lista que está dentro de uma tupla é possível sem gerar um erro no programa.

Embora a tupla seja comumente composta de dois ou mais elementos, podemos definir uma tupla vazia ou com apenas um elemento, por exemplo:



```
tuples.py
empty_tuple = ()
one_element_tuple = 'Apple',

print(type(empty_tuple))
print(type(one_element_tuple))

Shell
>>> %Run tuples.py
<class 'tuple'>
<class 'tuple'>
```


8.2 Dicionários

Nós já passamos por duas estruturas de dados até aqui, listas e tuplas. Em ambos os casos, as estruturas eram utilizadas para armazenar valores que descreviam quase a mesma ideia ou valores muito próximos conceitualmente. O dicionário, por sua vez, é uma estrutura de dados que nos permite uma flexibilidade maior de armazenamento de dados, e mais do que isso, um acesso diferenciado a estes dados.

Os dicionários que nós já conhecemos tem uma estrutura padrão, o par verbete e significado. Dicionários em Python são parecidos, mas chamaremos o nosso par de dados de chave e valor.

As nossas chaves serão sempre **strings**, e devem descrever o “significado”, ou o que representa o nosso valor. Separamos esse par por dois pontos “:”, e cada par chave/valor por uma vírgula “,”.

Assim como em listas, acessaremos os dados usando a notação de colchetes, mas ao invés de índices, informaremos a chave correspondente ao valor que queremos acessar:



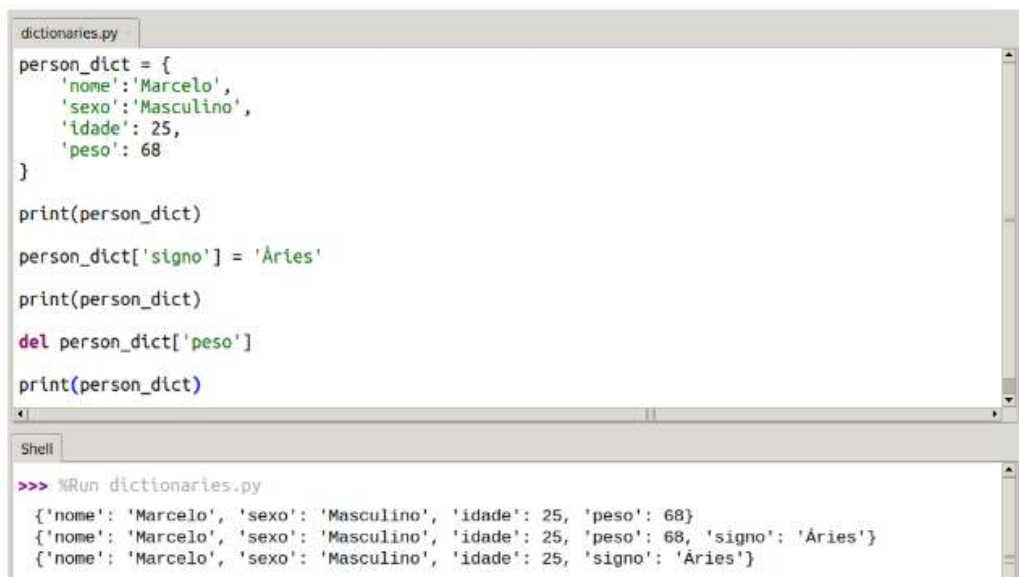
```
dictionary.py
person_dict = {
    'nome': 'Marcelo',
    'sexo': 'Masculino',
    'idade': 25,
    'peso': 68
}

print('Nome: {}'.format(person_dict['nome']))

Shell

>>> %Run dictionary.py
Nome: Marcelo
```

O dicionário possui alta mutabilidade, permitindo alteração direta de um valor apenas referenciando-o pela chave correspondente, a adição de um par chave/valor de maneira dinâmica, ou a deleção de um par usando a função **del**.



```
dictionary.py
person_dict = {
    'nome': 'Marcelo',
    'sexo': 'Masculino',
    'idade': 25,
    'peso': 68
}

print(person_dict)

person_dict['signo'] = 'Áries'

print(person_dict)

del person_dict['peso']

print(person_dict)

Shell

>>> %Run dictionary.py
{'nome': 'Marcelo', 'sexo': 'Masculino', 'idade': 25, 'peso': 68}
{'nome': 'Marcelo', 'sexo': 'Masculino', 'idade': 25, 'peso': 68, 'signo': 'Áries'}
{'nome': 'Marcelo', 'sexo': 'Masculino', 'idade': 25, 'signo': 'Áries'}
```

8.2.1 Dicionários

Ao usarmos o **For Loop** com um dicionário, a variável definida logo após o **for**, representará os índices (chaves) presentes no dicionário, diferente das listas, em que essa variável representa os valores.

```
dictonaries.py
person_dict = {
    'nome': 'Marcelo',
    'sexo': 'Masculino',
    'idade': 25,
    'peso': 68
}

for person in person_dict:
    print('{}: {}'.format(person, person_dict[person]))
```

Shell

Python 3.6.7

```
>>> %Run dictonaries.py
nome: Marcelo
sexo: Masculino
idade: 25
peso: 68
```

8.2.2 Métodos de dicionário

Caso você queira iterar sobre os valores presentes num dicionário, pode se utilizar do método **.values()** que retorna uma lista especial (do tipo `dict_values`) composta apenas pelos valores que compõem esse dicionário, desta forma:

```
dictonaries.py
person_dict = {
    'nome': 'Marcelo',
    'sexo': 'Masculino',
    'idade': 25,
    'peso': 68
}

for value in person_dict.values():
    print(value)
```

Shell

Python 3.6.7

```
>>> %Run dictonaries.py
Marcelo
Masculino
25
68
```

Assim como o método **.values()**, também temos o método **.keys()** que retorna todas as chaves de um dicionário, e o método **.items()** que retorna uma lista com tuplas compostas dos pares chave/valor de um dicionário.

```
Shell
>>> person_dict.keys()
dict_keys(['nome', 'sexo', 'idade', 'peso'])
>>> person_dict.items()
dict_items([('nome', 'Marcelo'), ('sexo', 'Masculino'), ('idade', 25), ('peso', 68)])
```

O tipo dicionário também utiliza alguns dos métodos de lista, como **clear()**, **pop()** e **copy()**.

Nós sabemos que podemos acessar um elemento de um dicionário usando diretamente a sua chave correspondente em notação de colchetes. No entanto, caso tentemos acessar um elemento desta maneira e a chave que utilizamos tiver sido excluída, ou simplesmente não tiver sido adicionada naquele dicionário específico, o interpretador do Python irá retornar um erro e logo, quebrar a execução do programa. Por isso, é mais recomendado que sempre que quisermos recuperar o valor dentro de um dicionário, a partir de sua chave usemos o método **.get()**, passando a chave correspondente para que o programa retorne o valor, se ele existir, ou retorne `None`, se ele não existir, permitindo que nós tratemos esse caso mais facilmente.

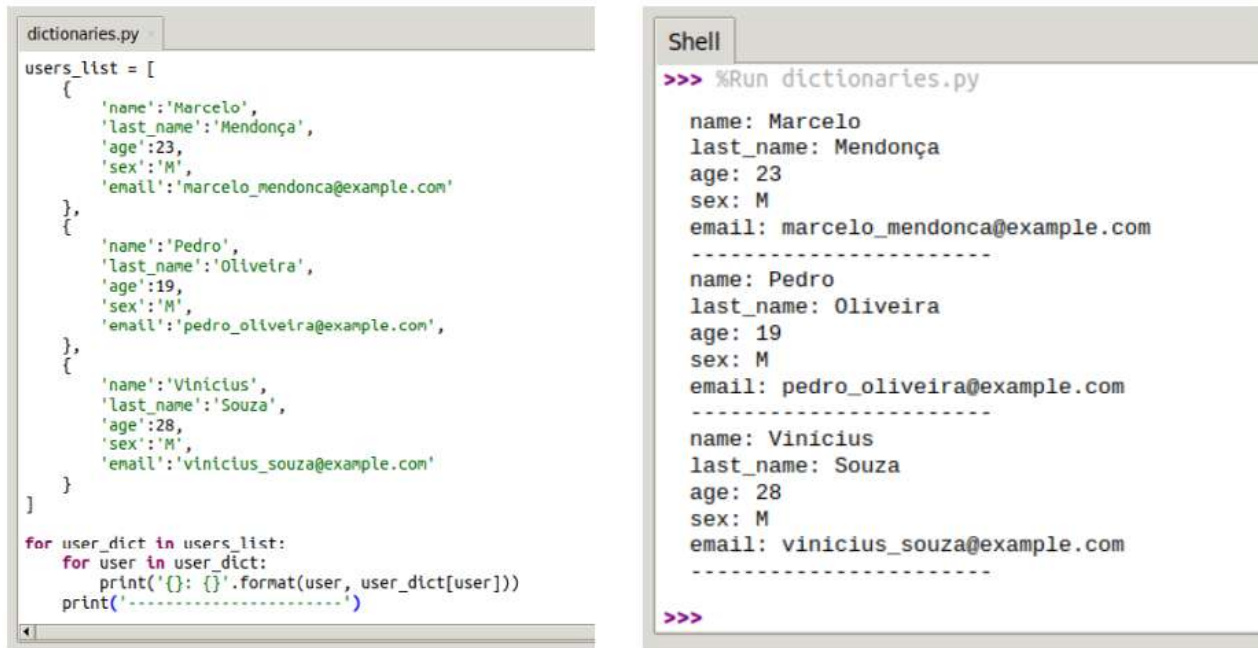
```
dictonaries.py
user_dict = {
    'name': 'Marcelo',
    'last_name': 'Mendonça',
    'email': 'marcelo_mendonca@example.com'
}

if user_dict.get('name'):
    print('Hey, {}! How are you doing?'.format(user_dict.get('name')))
else:
    print('Hey! How are you doing?')
```

Neste momento, você começa a ter cada vez mais noção do poder que estamos ganhando com a linguagem Python. Assim como no nosso exemplo, dicionários podem ser usados para guardar dados de uma pessoa, de objetos, ou de produtos mais generalizados, se pensarmos no sistema de

uma loja, por exemplo.

Em todos esses casos, no entanto, falamos sempre de forma pluralizada sobre o que iremos tratar, ou seja, o mais comum é na verdade que tenhamos sempre um conjunto de dicionários que guarda dados de diferentes objetos, e isso nós já sabemos como fazer. Assim como todos os outros tipos de variáveis que já vimos, os dicionários também podem ser armazenados em listas:



```
dictionaries.py
users_list = [
    {
        'name': 'Marcelo',
        'last_name': 'Mendonça',
        'age': 23,
        'sex': 'M',
        'email': 'marcelo_mendonca@example.com'
    },
    {
        'name': 'Pedro',
        'last_name': 'Oliveira',
        'age': 19,
        'sex': 'M',
        'email': 'pedro_oliveira@example.com',
    },
    {
        'name': 'Vinicius',
        'last_name': 'Souza',
        'age': 28,
        'sex': 'M',
        'email': 'vinicius_souza@example.com'
    }
]

for user_dict in users_list:
    for user in user_dict:
        print('{}: {}'.format(user, user_dict[user]))
        print('-----')
```

```
Shell
>>> %Run dictionaries.py

name: Marcelo
last_name: Mendonça
age: 23
sex: M
email: marcelo_mendonca@example.com
-----
name: Pedro
last_name: Oliveira
age: 19
sex: M
email: pedro_oliveira@example.com
-----
name: Vinicius
last_name: Souza
age: 28
sex: M
email: vinicius_souza@example.com
-----
>>>
```

Repare que nesse exemplo, usamos algo que ainda não tinha aparecido no nosso código, embora seja até comum, o chamado **laço(loop) aninhado**, mais popularmente conhecido como **nested loop**. Toda vez que temos objetos iteráveis (**strings**, listas, tuplas, dicionários) dentro de outros, é possível que nós precisemos dentro do **loop** que itera sobre o objeto mais externo, fazer uma repetição também sobre esse iterável interno.

No exemplo acima, o iterável externo foi a lista, enquanto o interno, cada dicionário que a compôs. Não há um limite para quantos **nested loops** você pode utilizar, se dentro de cada dicionário tivesse uma chave de “filhos”, por exemplo, guardando o nome dos filhos de cada usuário, você poderia também iterar sobre essa lista interna.

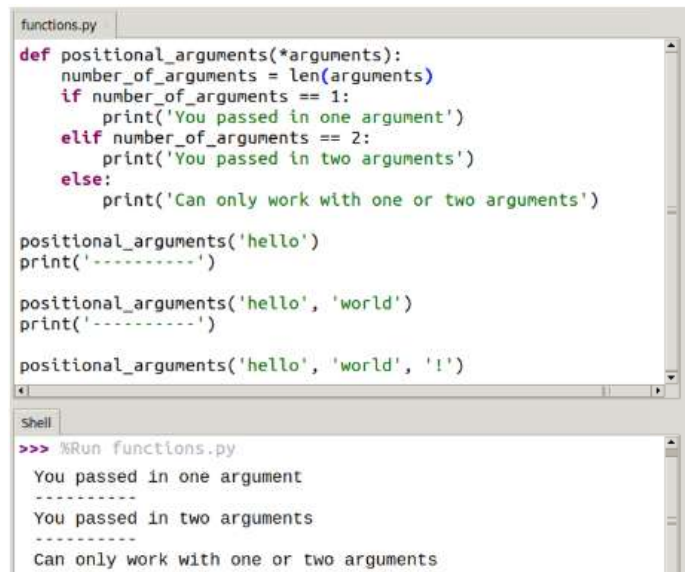
8.3 Quantidade arbitrária de argumentos ***args**, ****kwargs**

Agora com o conceito de dicionários e tuplas em mãos, vamos voltar a falar um pouco sobre funções e ganhar mais algumas opções para passagem de parâmetros.

Algumas funções podem ter um número variável de argumentos, dependendo

do momento de uso, assim como a função **range()**, estudada no capítulo sobre **For Loop**.

Se colocarmos um asterisco “*” na frente de um argumento, ele recebe como valor uma tupla de todos os parâmetros posicionais que forem passados:



```
functions.py
def positional_arguments(*arguments):
    number_of_arguments = len(arguments)
    if number_of_arguments == 1:
        print('You passed in one argument')
    elif number_of_arguments == 2:
        print('You passed in two arguments')
    else:
        print('Can only work with one or two arguments')

positional_arguments('hello')
print('-----')

positional_arguments('hello', 'world')
print('-----')

positional_arguments('hello', 'world', 'I')

Shell
>>> %Run functions.py
You passed in one argument
-----
You passed in two arguments
-----
Can only work with one or two arguments
```

Geralmente essas funções são declaradas fazendo um controle sobre quantos argumentos foram passados para retornarem respostas diferentes, utilizando a função **len()** para tuplas.

De forma similar, se colocarmos dois asteriscos “**” na frente de um argumento, ele recebe como valor todas as combinações chave/valor que forem passadas. Veja na prática:



```
functions.py
def keyword_arguments(**kwargs):
    for keyword in kwargs:
        print('{}: {}'.format(keyword, kwargs.get(keyword)))
    print('-----')

keyword_arguments(name='Felipe', sex='M')

keyword_arguments(language='Python', position_in_ranking='1st')

Shell
>>> %Run functions.py
name: Felipe
sex: M
-----
language: Python
position_in_ranking: 1st
-----
```

Obs: Embora seja muito comum declararmos os parâmetros de uma função que recebe keyword arguments (kwargs), com esse nome, você pode escolher o nome que quiser se for facilitar a sua leitura do código.

The background of the entire page is a dark blue field with a complex, light blue circuit-like pattern. This pattern consists of numerous thin, interconnected lines and small circular nodes, resembling a printed circuit board or a neural network diagram. The lines and nodes are more densely packed on the left side and become sparser towards the right.

CAPÍTULO

09

PROGRAMAÇÃO
ORIENTADA A
OBJETOS

9.1 Introdução ao paradigma

O termo "Programação Orientada a Objetos" é antigo, do final da década de 1960. Linguagens de programação mais básicas eram (e ainda são) estruturadas como uma sequência de comandos, fazendo com que muitas vezes seja difícil distinguir cada parte de um todo, conforme o código e a complexidade vão aumentando.

Desta forma, pensar em estruturar ou agrupar um conjunto de variáveis e funções que dizem respeito a um conceito, criando blocos conceituais reutilizáveis ao longo da programação, é o que dá origem a esta forma de estrutura de software.

Antes de falarmos de objetos, precisamos entender o que é uma classe. Nós vimos há pouco o conceito de dicionários e como eles podem ser usados para guardar valores que seguem um "padrão". Muita atenção nestas aspas, afinal, sabemos que dicionários são altamente mutáveis, então esse suposto "padrão" depende apenas do programador respeitá-lo ou não.

As classes, tem um uso parecido, entendemos uma classe como um modelo, uma estrutura padronizada que representa um objeto do mundo real (ou uma necessidade recorrente do sistema). Toda vez que utilizamos a função **type()**, ao longo do curso, você deve ter percebido que o tipo era sempre precedido pela palavra "class". Pois bem, todos os tipos e estruturas de dados que estudamos até aqui são classes, cada um com o seu padrão – diferentes maneiras sobre como podemos operar os dados e como eles se comportam no programa.

As classes são modeladas a partir de dois conceitos básicos que pode – ou deve – estar presente.



Um destes, são os atributos de um objeto. Atributos são como características desse nosso modelo, por exemplo, se modelarmos o usuário de um programa – generalizando o máximo possível – teríamos atributos como: nome, sobrenome, gênero, data de nascimento, entre outros.



O segundo dos conceito é algo que nós na verdade já estudamos aqui, que é o conceito de métodos. Os métodos são operações, comportamentos que toda **instância** (vamos entender melhor sobre isso em um momento) de uma classe poderá realizar. Ao estudarmos **strings**, por exemplo, vimos métodos que só podem ser usados para este tipo de dados, como o **.upper()** por exemplo. Seria inconcebível aplicar este método para a classe que modela o tipo **int** ou **float** por exemplo.

As **instâncias** de uma classe que acabamos de mencionar, podem ser chamados os **objetos**. Objetos nada mais são do que criações feitas a partir de um modelo previamente criado. É na classe que definimos quais serão os atributos e como eles serão usados, mas é em um objeto que esses atributos ganham um valor. Cada instância de uma classe usuário, por exemplo, terá um nome, sobrenome, gênero e data de nascimento com um valor diferente.

9.2 Criando uma classe

Já sabemos que o Python é todo estruturado em classes, desde seus tipos primitivos de dados, agora é hora de criarmos nossas próprias classes personalizadas. A sintaxe para definir uma classe simples, é também, muito simples, veja abaixo:

```
classes.py
class User():
    name = ""
    last_name = ""
    sex = ""
```

No exemplo acima, *name*, *last_name* e *gender* são os **atributos** da nossa classe, que podem ter um valor padrão, ou não. No nosso exemplo, cada atributo recebeu uma **string** vazia, mas poderíamos também ter passado o objeto nulo (**None**), se preferíssemos.

Como foi dito, normalmente são os objetos que possuem os reais valores. Então vamos instanciar um objeto da classe *User* e dar valores reais aos atributos:

```
classes.py
class User():
    name = None
    last_name = None
    sex = None

usuario = User()
usuario.name = "Marcelo"
usuario.last_name = "Mendonça"
usuario.sex = "M"

print("Name: {0.name} {0.last_name}; Sex: {0.sex}".format(usuario))

Shell
>>> %Run classes.py
Name: Marcelo Mendonça; Sex: M
```

Um objeto é instanciado de maneira muito similar à declaração de uma variável de tipo primitivo, a diferença é que quando declaramos variáveis

“comuns” o interpretador do Python é capaz de inferir o tipo que estamos declarando, então não precisamos especificar. No caso dos objetos, a primeira linha após a declaração da classe é dedicada basicamente a informar ao interpretador que a variável *usuario* deve ser identificada como um objeto da classe *User*. Depois desse momento, podemos acessar e alterar os atributos do objeto diretamente, usando a notação de ponto que já conhecemos.

Sabemos, no entanto, que uma classe não é normalmente composta apenas de atributos, mas também de seus métodos. Vamos portanto adicionar um método de apresentação à nossa classe *User*:

```
classes.py
class User():
    name = None
    last_name = None
    sex = None

    def introduceMyself(self):
        print("My name is {0.name} {0.last_name}, nice to meet you!".format(self))

usuario = User()
usuario.name = "Marcelo"
usuario.last_name = "Mendonça"
usuario.sex = "M"

usuario.introduceMyself()

Shell
>>> %Run classes.py
My name is Marcelo Mendonça, nice to meet you!
```

A declaração de um método de uma classe é quase idêntica a de uma função, exceto por um detalhe – o parâmetro **self**. O parâmetro **self** é como uma cópia do nosso objeto que é passada ao método para que possamos acessar seus atributos e sem o qual o nosso método não pode ser definido. Portanto o parâmetro **self** é obrigatório a todo método de classe em Python.

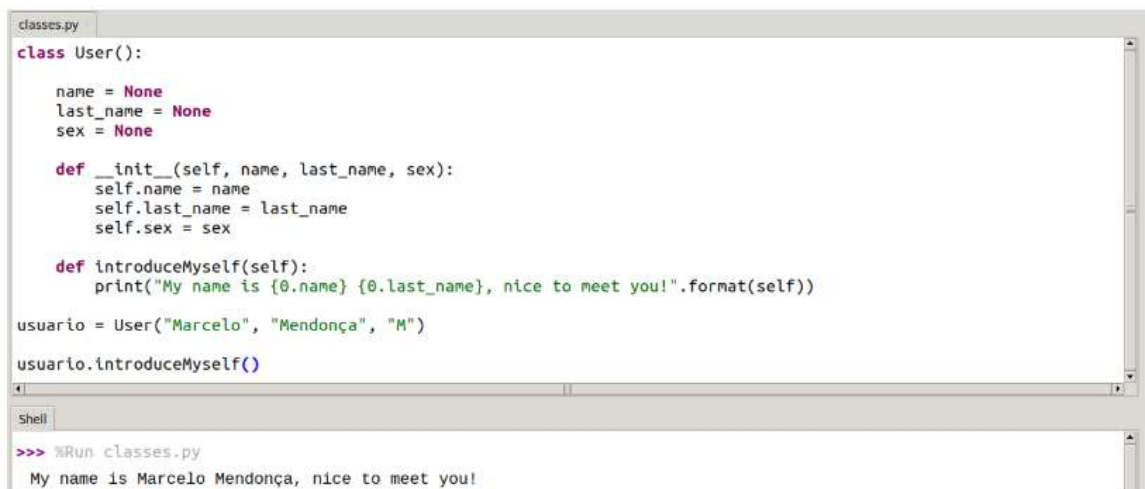
9.3 Métodos Mágicos

As classes em Python podem conter métodos especiais, com sintaxe de declaração padrão, chamados **métodos mágicos**. Os

métodos mágicos podem ser usados para conversão (casting) de um objeto para um tipo específico ou para definição do seu comportamento mediante alguma operação.

Os métodos mágicos, diferente dos demais, não precisam ser chamados explicitamente para o objeto em questão. Estes métodos são naturalmente executados pelo Python, de acordo com algumas operações a que o objeto é submetido.

O mais comum e provavelmente o mais importante dentre eles, é o que chamamos de **método construtor**. A operação que dispara o uso do método construtor é a própria instanciação ou inicialização do objeto, justificando o nome. Normalmente, utilizamos este método para dar os valores iniciais dos atributos de uma classe – até porque, fazer isso linha a linha pode ser muito exaustivo:



```
class User():
    name = None
    last_name = None
    sex = None

    def __init__(self, name, last_name, sex):
        self.name = name
        self.last_name = last_name
        self.sex = sex

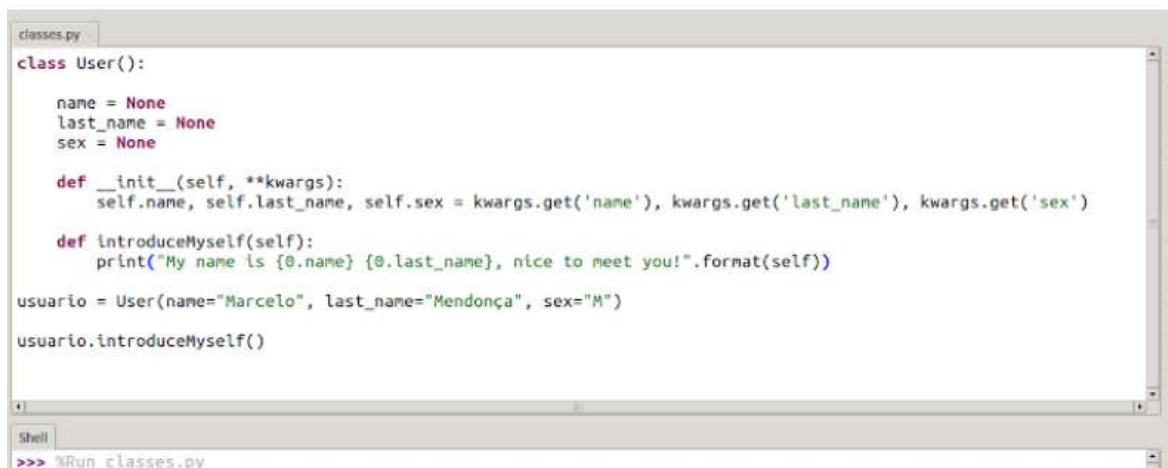
    def introduceMyself(self):
        print("My name is {0.name} {0.last_name}, nice to meet you!".format(self))

usuario = User("Marcelo", "Mendonça", "M")
usuario.introduceMyself()
```

```
>>> %Run classes.py
My name is Marcelo Mendonça, nice to meet you!
```

Repare que os parâmetros do método construtor (**__init__**) são passados no momento da instância da classe, afinal, é nesse momento que o método é chamado pelo programa.

Você pode sofisticar o seu método **__init__** utilizando os conceitos de keyword arguments (argumentos chave/valor):



```
class User():
    name = None
    last_name = None
    sex = None


    def __init__(self, **kwargs):
        self.name, self.last_name, self.sex = kwargs.get('name'), kwargs.get('last_name'), kwargs.get('sex')

    def introduceMyself(self):
        print("My name is {0.name} {0.last_name}, nice to meet you!".format(self))

usuario = User(name="Marcelo", last_name="Mendonça", sex="M")
usuario.introduceMyself()
```

```
>>> %Run classes.py
```


Ou argumentos posicionais:



```
classes.py
class User():
    name = None
    last_name = None
    sex = None

    def __init__(self, *args):
        self.name, self.last_name, self.sex = args

    def introduceMyself(self):
        print("My name is {0.name} {0.last_name}, nice to meet you!".format(self))

usuario = User("Marcelo", "Mendonça", "M")
usuario.introduceMyself()

Shell
>>> %Run classes.py
My name is Marcelo Mendonça, nice to meet you!
```

Além do método construtor, existem vários outros métodos mágicos que podem ser muito úteis na construção do seu algoritmo. Veja:

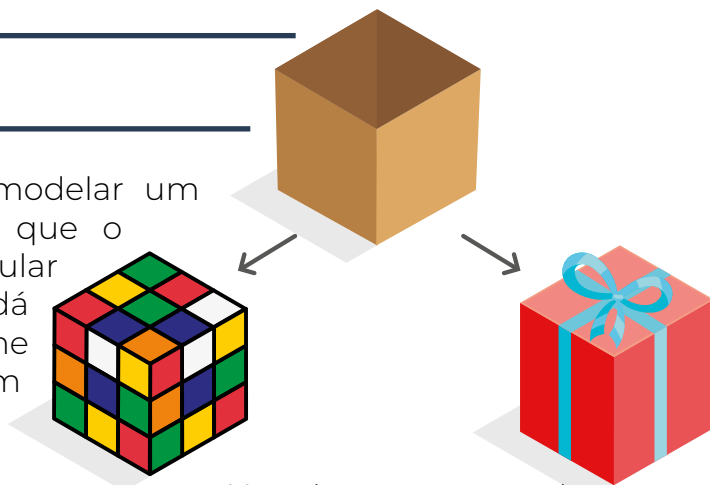
| | |
|---|---|
| <code>object.__lt__(self, other)</code> | Deve retornar um valor que represente o objeto numa operação de “menor que” (<) |
| <code>object.__le__(self, other)</code> | Deve retornar um valor que represente o objeto numa operação de “menor ou igual a” (<=) |
| <code>object.__eq__(self, other)</code> | Deve retornar um valor que represente o objeto numa comparação de igualdade (==) |
| <code>object.__ne__(self, other)</code> | Deve retornar um valor que represente o objeto numa comparação de “diferente de” (!=) |
| <code>object.__gt__(self, other)</code> | Deve retornar um valor que represente o objeto numa operação de “maior que” (>) |
| <code>object.__ge__(self, other)</code> | Deve retornar um valor que represente o objeto numa operação de “maior ou igual a” (>=) |
| <code>object.__add__(self, other)</code> | Deve retornar um valor que represente o objeto numa operação de adição (+) |
| <code>object.__sub__(self, other)</code> | Deve retornar um valor que represente o objeto numa operação de subtração (-) |
| <code>object.__mul__(self, other)</code> | Deve retornar um valor que represente o objeto numa operação de multiplicação (*) |
| <code>object.__truediv__(self, other)</code> | Deve retornar um valor que represente o objeto numa operação de divisão (/) |
| <code>object.__floordiv__(self, other)</code> | Deve retornar um valor que represente o objeto numa operação de divisão inteira (//) |
| <code>object.__mod__(self, other)</code> | Deve retornar um valor que represente o objeto numa operação de módulo (%) |
| <code>object.__pow__(self, other)</code> | Deve retornar um valor que represente o objeto numa operação de potenciação (**) |
| <code>object.__int__(self)</code> | Deve retornar um valor que represente o objeto numa possível conversão para inteiro |
| <code>object.__float__(self)</code> | Deve retornar um valor que represente o objeto numa possível conversão para número real |
| <code>object.__str__(self)</code> | Deve retornar um valor que represente o objeto numa possível conversão para uma string |

Para exemplificar o uso ou a chamada destes métodos, vamos olhar o método `__str__`, que é chamado quando uma função que pede um valor string – ou quando fazemos o casting desse objeto para string – é chamada com o nosso objeto como parâmetro:

Repare que no exemplo acima, a função `print()` é usada diretamente sobre o objeto `usuario`, normalmente, o Python identificaria que `usuario` não é um tipo `string`, e retornaria um erro. Só que, a partir da declaração do método `__str__`, nós “ensinamos” ao nosso objeto como responder caso ele precise se comportar como um valor textual.

9.4 Herança

Falamos sobre o uso de classes para modelar um determinado objeto do nosso sistema, e que o paradigma de POO tornou-se tão popular justamente pela possibilidade que este nos dá de fácil reuso de código. Pois bem, agora imagine um sistema para gestão empresarial de um supermercado, por exemplo. Um sistema desse tipo deve ser capaz de gerenciar todos os funcionários, mas como você certamente já sabe, o corpo de funcionários de um supermercado é dividido em vários cargos: operador de caixa, gerente de caixa, empacotador, gerente de compras, etc.



Embora esses funcionários tenham funções diferentes entre si, há características e funções que são comuns a todos. Como características, podemos citar: nome, sobrenome, idade, gênero, CPF, cargo, salário, (... e como funções: bater o ponto, solicitar férias, aumento de salário, e muitas outras. Ao mesmo tempo, existem características e funções específicas de cada cargo. Então, será que é necessário escrever uma classe para cada “tipo” de funcionário que esse sistema irá gerenciar? Onde está a vantagem de reuso de código da Programação Orientada a Objetos nisso?

Pois bem, felizmente, há uma saída para esta situação. É aqui que entra o conceito de herança. A herança é a definição de uma classe “primitiva”, a partir da qual se podem escrever classes específicas em que serão definidos apenas os atributos e métodos reservados desta classe. Isto é, enquanto a classe `Funcionário`, terá métodos como os já discutidos, a classe `OperadorDeCaixa` terá um método específico como `finalizar_compra()`, ou `solicitar_cancelamento()`, que a classe `Empacotador`, por exemplo, jamais terá.

A declaração da relação de herança entre classes em Python é muito simples. Basta no momento da declaração da classe “filha” (classe que herda), passar o nome da classe “mãe” (classe que é herdada) entre os parênteses que seguem o nome da

classe:

```
inheritance.py

class Employee(object):
    name, last_name, sex, age, salary, role = None, None, None, None, None, None
    def __init__(self, **kwargs):
        self.name, self.last_name, self.sex, self.age, self.salary, self.role = tuple(kwargs.values())

class Manager(Employee):
    def __init__(self, **kwargs):
        kwargs['role'] = "Manager";
        super().__init__(**kwargs)

    def promote_employee(self, employee, new_role):
        employee.role = new_role
        print("{0.name} was promoted to {0.role} by {1.name}".format(employee, self))

class Packer(Employee):
    def __init__(self, **kwargs):
        kwargs['role'] = "Packer";
        super().__init__(**kwargs)

manager = Manager(name="Nataly", last_name="Evans", sex="F", age=32, salary=4000)
packer = Packer(name="Tag", last_name="Jones", sex="M", age=23, salary=2000)
manager.promote_employee(packer, "Supply Manager")

Shell

>>> %Run inheritance.py
Tag was promoted to Supply Manager by Nataly
```

No exemplo acima possuímos três classes, duas classes filhas (Manager e Packer) e a classe mãe (Employee), neste caso, tudo que a classe mãe faz é a definição dos atributos em comum através do seu método construtor. A classe Manager, no entanto, define um método específico, que é a capacidade de promover um funcionário com o método `promote_employee()`.

Além disso, ambas as classes filhas fazem uso de um recurso muito importante da relação de herança, que é a sobrescrita de método ou **overwriting**. Repare que ambas possuem sua própria implementação do método construtor, cada uma atribuindo um valor padrão para a função (role) do funcionário em questão. No entanto, logo após a definição desse atributo, ambos os métodos chamam o construtor da classe mãe através do método **super()**. Este método é utilizado para fazer referência à classe mãe, seja para acessar um atributo ou um método da mesma.

Obs: É importante ressaltar que a sobrescrita de um método na classe filha não precisa ter qualquer relação com o respectivo método na classe mãe.

The background of the entire page is a dark blue field filled with a complex, abstract pattern of light blue lines. These lines, resembling a circuit board or a neural network, branch out and connect at various points, some ending in small circular nodes. The pattern is dense and covers the entire area, creating a technical and digital aesthetic.

CAPÍTULO

10

APLICAÇÕES
GUI



10.1 Aplicações GUI com Tkinter

GUI é a sigla para Graphical User Interface, ou, Interface Gráfica do Usuário. Até aqui, toda a execução dos nossos programas se dava diretamente pelo shell do terminal. A entrada de dados por parte do usuário era sempre feita com a função **input()**, mas um produto final não pode seguir esse modelo por razões óbvias. Portanto, neste capítulo começaremos a estudar o desenvolvimento de uma aplicação Desktop, a criação de janelas gráficas que sejam mais “amigáveis” ao usuário e claro, de mais fácil interação.

A instalação do Python carrega como padrão – exceto em algumas distribuições Linux – uma biblioteca responsável pela criação dessas aplicações, chamada Tkinter. Essa biblioteca conta com uma série de classes que podem ser usados para criação de janelas e seus componentes – dos mais comuns aos mais completos – como botões, caixas de texto, imagens, entre outros.

Antes de começar os exemplos, é importante apresentar alguns conceitos básicos com que a biblioteca trabalha, e que na verdade são muito comuns no desenvolvimento de interfaces.

O primeiro deles, é o contêiner. O contêiner nada mais é do que uma espécie de retângulo para guardar componentes mais específicos do nosso layout, ou até mesmo outros contêineres.

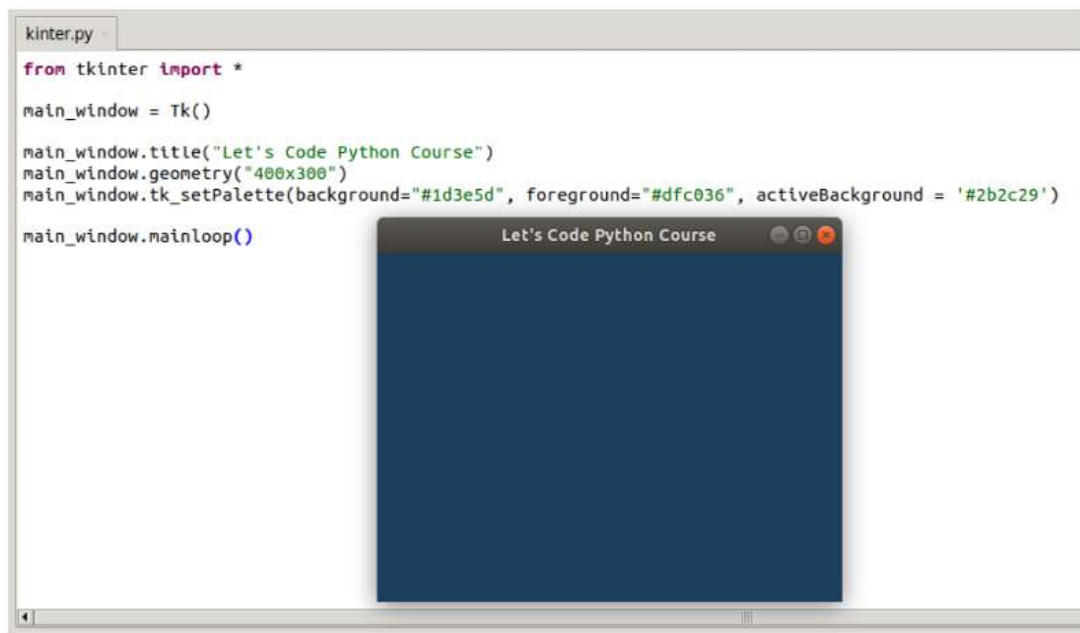
Aliado aos contêineres, temos os widgets. Os widgets são os componentes com que o usuário irá interagir diretamente. O conceito por trás dos botões, caixas de textos e menus de seleção, são exemplos de widgets.

Até aqui, falamos apenas da parte visual das interfaces. Ó que já sabemos que um programa funcional não se faz apenas com interfaces, por isso vamos agora falar das interações. As interações do usuário com os widgets, serão gerenciadas pelos chamados Event Handlers, funções que serão executadas a partir de interações específicas do usuário em cada widget.

Os Event Handlers, no entanto, não trabalham sozinhos nessa parte funcional do nosso programa. Para que um Event Handler seja chamado, é preciso que haja um Event Loop, que são aqueles que têm o papel de “observar” as ações do usuário para designar o Event Handler responsável. Com esses pontos explicados, vamos então começar.

10.2 Definindo uma janela

As janelas definidas usando Tkinter são instâncias da classe Tk do módulo, a partir das quais nós podemos definir propriedades como título, tamanho e cores da nossa janela:



A primeira coisa a se fazer para utilizar a biblioteca Tkinter é importá-la. Para não precisarmos referenciar a biblioteca toda vez que tivermos de utilizar um de seus métodos ou componentes, importamos usando o **from**.

Depois disso, basta instanciar a classe e definir alguns atributos, como o título, através do método **title()**, o tamanho, com o método **geometry()**, que recebe os parâmetros de largura e altura em pixels, separados por um x; e ainda as cores padrão usadas na janela, e seus componentes com **tk_setPalette()**.

Existem três formas de alocar widgets nas janelas (e contêineres) usando o Tkinter: **pack()**, **grid()** e **place()**. O método **pack()** é provavelmente o mais simples dos três. Se usado sem qualquer parâmetro, ele apenas aloca o elemento no contêiner definido como "pai" do elemento, por padrão no topo do contêiner e centralizado.





Para demonstrar o uso do método **pack()**, utilizamos um botão da classe Button() do Tkinter. Os parâmetros de declaração de boa parte dos widgets seguem o padrão acima, embora os parâmetros text e bg não sejam obrigatórios. O primeiro parâmetro que passamos é sempre o contêiner a que o widget pertence, nesse caso, a própria janela.

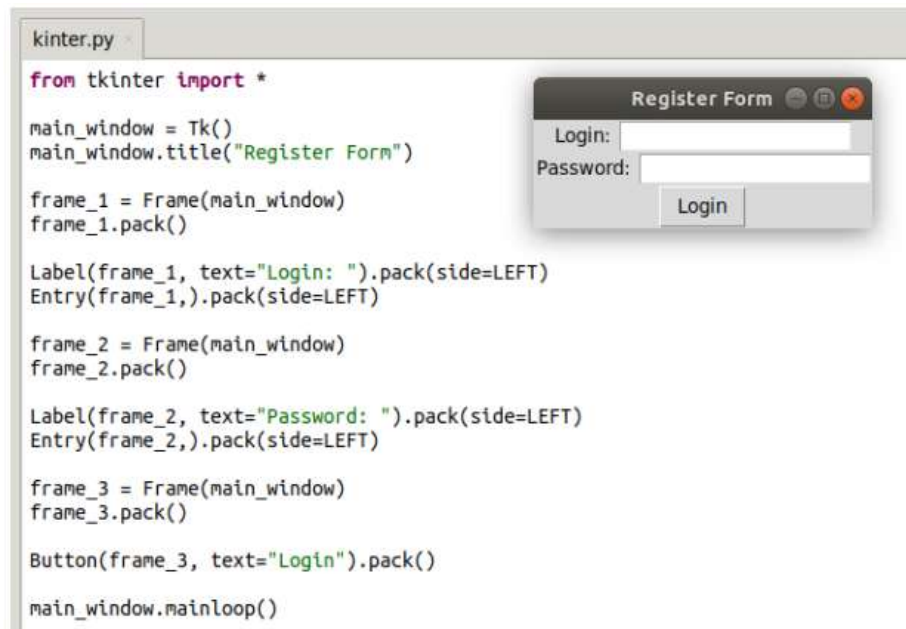
O método de alocação **pack()** aceita alguns parâmetros para posicionamento dos elementos, tais quais:

- **fill**: este parâmetro pode receber como valor as constantes X, Y, ou BOTH. Indica se o elemento que está sendo posicionado deve preencher a extensão horizontal, vertical ou ambas, respectivamente, do contêiner a que pertence.
- **side**: este parâmetro aceita as constantes LEFT, RIGHT, TOP e BOTTOM, ou as strings “left”, “right”, “top”, e “bottom”, correspondentes, usadas para definir uma “âncora” para o nosso elemento. Toda vez que possuímos mais de um elemento na mesma âncora – - LEFT, por exemplo -, é como se disséssemos a cada elemento que vai sendo posicionado, para estar o mais à esquerda possível, sem sobrepor o elemento já posicionado, claro.
- **expand**: este parâmetro recebe um valor booleano que indica se o widget pode ou não expandir para ocupar o valor disponível no elemento externo. Por padrão, os elementos ancorados em TOP ou BOTTOM são permitidos expandir horizontalmente se fill=X, e os ancorados em LEFT ou RIGHT podem expandir verticalmente se fill=Y, mesmo com expand=false. No entanto, se este valor for True, não importando a âncora, o elemento será capaz de expandir para qualquer que seja o sentido especificado em fill.
- **padx**, **pady**: estes parâmetros determinam o espaçamento externo entre widgets na horizontal/vertical.
- **ipadx**, **ipady**: estes parâmetros determinam o espaçamento interno dos widgets na horizontal/vertical.

É importante ressaltar que esta abordagem não é a mais comum, ou mesmo a mais funcional, de dispor os elementos. Ao invés de incluí-los diretamente na janela principal, como demonstrado, o mais comum seria utilizar um contêiner interno para abrigá-los. Para isso, usamos os Frames. Os Frames, são também exemplos de widgets, mas não possuem função similar aos demais. Os Frames são usados apenas para abrigar outros elementos, como retângulos invisíveis (ou em alguns casos, visíveis) para ajudar na organização dos elementos.

Repare que como foi omitido o valor de **side** para os Frames e Buttons, estes foram posicionados centralizados ao topo do elemento mais externo. No caso dos Frames, como há concorrência de âncoras no contêiner externo, ganha prioridade o primeiro a ser declarado.

Além dos botões, outros widgets padrão podem ser utilizados nas nossas telas, como os Labels, que são textos simples, normalmente usados em conjunto com um campo de entrada de dados, que pode ser uma caixa de texto, de seleção, entre outras opções. Veja abaixo:



Repare que organizar o layout se utilizando apenas do método `pack()` [negrito] acaba sendo extremamente cansativo e nada intuitivo. Vamos apresentar então o próximo método de organização, que é o **grid()**. Este método nos permite organizar os widgets em formato de tabela, informando a cada momento, a linha (row) e coluna (column) em que deve estar presente o elemento que estamos alocando.

Obs: A contagem das linhas e colunas inicia em 0, e no caso de não ter sido passado nenhum valor para uma linha ou coluna, o método entende que o valor deve ser 0.

Além dos parâmetros `row` e `column`, é muito comum o uso dos parâmetros **columnspan** e **rowspan**. Estes parâmetros permitem que um determinado elemento ocupe mais de uma coluna (`columnspan`) e/ou mais de uma linha (`rowspan`):



Saindo um pouco do campo apenas visual, vamos começar a entender como podemos tratar eventos ou disparar funções de acordo com a interação com alguns elementos da tela.


Primeiramente, vamos falar sobre o botão. Afinal, todo mundo espera algum retorno a partir do clique de um botão. Quando declaramos um **Button()** do Tkinter, há um parâmetro que ainda não exploramos, o **command**. Esse parâmetro recebe uma função que será executada quando o botão for clicado:

```
kinter.py
from tkinter import *
main_window = Tk()
main_window.title("Let's Code Python Course")
frame = Frame(main_window)
frame.pack()

def callback():
    print("Clicked")

Button(frame, text="Trigger", command=callback).grid()

main_window.mainloop()
```



```
Shell
>>> %Run kinter.py
Clicked
```

Repare que é passado apenas o nome da função, sem os parênteses. Se você incluir os parênteses, o interpretador irá entender que você quer o retorno da função, e não apenas referenciá-la, como é o caso.

Tomemos como exemplo um formulário de cadastro em uma newsletter. Taremos os campos de nome, sobrenome e e-mail, e vamos armazenar em um dicionário – em uma aplicação real, provavelmente teríamos uma classe e guardaríamos esse valor em um objeto, para armazená-lo em um banco de dados ou arquivo local (csv, por exemplo).

```
kinter.py
main_window.title("Register Form")

frame = Frame(main_window)
frame.pack()

l_name = Label(frame, text="Name:")
l_name.grid()
e_name = Entry(frame)
e_name.grid(row=0, column=1)

l_lastname = Label(frame, text="Last name:")
l_lastname.grid(row=1)
e_lastname = Entry(frame)
e_lastname.grid(row=1, column=1)

l_email = Label(frame, text="E-mail:")
l_email.grid(row=2)
e_email = Entry(frame)
e_email.grid(row=2, column=1)

def convert_to_dict():
    user = dict()
    user['name'], user['last_name'], user['email'] = e_name.get(), e_lastname.get(), e_email.get()
    print(user)

Button(frame, text="Send", command=convert_to_dict).grid(row=3, column=1)
```



```
Shell
>>> %Run kinter.py
{'name': 'Pietro', 'last_name': 'Ribeiro', 'email': 'pietro@teste.com'}
```

Veja que utilizamos um novo método para os objetos **Entry**, o **.get()**, que como você pôde perceber, retorna o texto dentro desse campo de formulário.

Embora as ações atreladas aos botões sejam as mais comuns dentro de uma janela comum, você já deve saber que apenas executar funções ao clicar de um botão, pode não cobrir todas as necessidades de uma janela – tenha em mente que a biblioteca Tkinter pode ser usada para construir programas complexos e até jogos –. Existem vários outros tipos de eventos possíveis, mesmo para o mouse, como por exemplo podemos disparar eventos para o botão direito do mouse ou até com o botão de scroll (ou middle-button).

Para estender as nossas possibilidades com disparo de eventos, usaremos o método **bind()**. O método **bind** pode ser usado para qualquer um dos widgets que já estudamos até mesmo para frames.

Esse método recebe dois parâmetros, o evento que deve ser “percebido” e a função de callback, isto é, a função que deve ser executada quando aquele evento for disparado.

```
kinter.py
from tkinter import *

main_window = Tk()

main_window.title("Register Form")

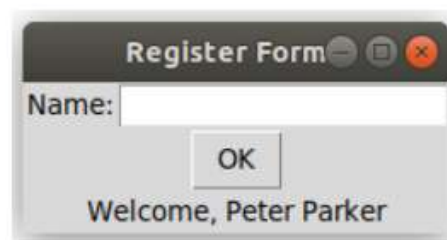
frame = Frame(main_window)
frame.pack()

l_name = Label(frame, text="Name:")
l_name.grid()
e_name = Entry(frame)
e_name.grid(row=0, column=1)

def show_welcome_message(event):
    l_welcome = Label(frame, text="Welcome, {}".format(e_name.get()))
    l_welcome.grid(row=2, columnspan=2)
    e_name.delete(0, END) # erases the content on an entry box text box

button = Button(frame, text="OK")
button.bind("<Button-1>", show_welcome_message)
button.grid(row=1, columnspan=2)

main_window.mainloop()
```



Os eventos passados para o método **bind**, possuem uma identificação específica, o evento no exemplo acima (**<Button-1>**) corresponde ao clique com o botão esquerdo do mouse, numa lógica parecida, o botão do meio e o direito correspondem aos eventos **<Button-2>** e **<Button-3>**, respectivamente. Segue abaixo uma tabela com os eventos possíveis para o método **bind**:



| | |
|-----------------|---|
| BX-Motion | Capta o movimento do mouse com o botão X pressionado |
| ButtonRelease-X | Capta a liberação do botão X após ter sido pressionado |
| Double-Button-X | Captura duplo clique com o botão X do mouse |
| Enter | O cursor do mouse está sobre no widget |
| Leave | Evento contrário ao <Enter>, disparado quando o mouse deixa o widget |
| FocusIn | Quando um determinado campo de digitação recebe o foco do teclado, como a partir da mudança de campos com a tecla TAB |
| FocusOut | Quando o widget perde o foco do teclado |
| FocusOut | A tecla Enter é pressionada |
| Key | A tecla key é pressionada, pode ser usada como: <BackSpace>, <Tab>, <CapsLock>, <Escape>, entre outros |
| a | A tecla a é pressionada. A grande maioria dos caracteres que podem ser impressos podem ser especificados dessa forma, com exceção do espaço (<space>) e do símbolo "menor que" (<less>) |

Repare que desta vez, a função **show_welcome_message()** recebe o parâmetro **event**, Toda função ligada a um evento, usando o método **bind**, deve receber este parâmetro. O valor **event** é um objeto que carrega atributos que podem ser úteis para a função, como:

| | |
|----------------|---|
| widget | Componente em que foi disparado o evento |
| x, y | Coordenadas do mouse no momento de disparo do evento |
| x_root, y_root | Coordenadas do mouse relativas ao canto superior esquerdo da tela |
| char | O código do caractere pressionado (em caso de eventos de teclado) |
| num | O número (1, 2 ou 3) do botão do mouse clicado (apenas para eventos de mouse) |
| type | O tipo do evento |

Existem vários métodos de widgets que você pode utilizar na construção da sua aplicação, além de um simples `.get()` para recuperar o valor de uma entrada, para uma referência completa sobre widgets, você pode acessar [aqui](#) a documentação do Python sobre o assunto.

The background of the entire page is a dark blue field filled with a complex, abstract pattern of light blue lines. These lines, resembling a circuit board or a network diagram, meander and branch out across the frame. Small, solid blue dots are placed at various points along these lines, adding to the technical and digital aesthetic of the design.

CAPÍTULO

11

REQUISIÇÕES
E APIS

11.1 Introdução a APIs e Requisições Web



As APIs (Application Programming Interfaces) são interfaces de programação que permitem a um programa uma comunicação de dados (trazendo ou enviando) usando requisições via internet.

Normalmente se utiliza uma API como uma funcionalidade complementar ao objetivo final do programa, como por exemplo uma API para cálculo de frete em uma loja virtual, ou uma feature que informa o clima em um blog.

É importante que neste momento tomemos cuidado para não fazer uma confusão com o termo interface.. No caso das APIs, embora o termo apareça na própria definição, não estamos falando de uma interface gráfica, mas sim, de um meio de interação que possibilita o envio e recebimento de dados, a partir de alguns parâmetros.

Existem inúmeras APIs disponíveis no mercado para os mais variados fins, seja para dados climáticos — como o WeatherAPI da Yahoo —, financeiros — como a Alpha Advantage —, geográficos — como o Google Maps API —, entre tantos outros.

11.2 APIs em Python com a biblioteca Requests

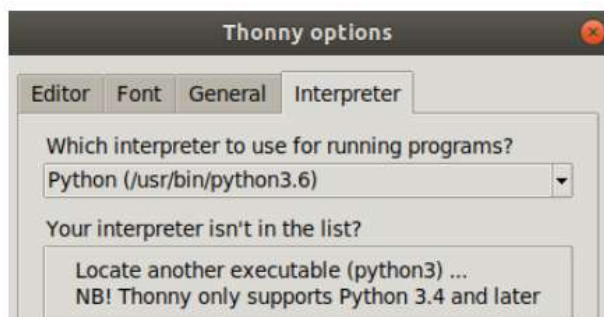
Existe uma biblioteca em Python que lida com requisições web de maneira fácil e intuitiva, chamada **requests**.

Diferente da biblioteca tkinter que

estudamos há pouco, a biblioteca `requests` não é padrão da linguagem, portanto, precisa ser instalada. A instalação desta biblioteca é muito simples, basta usar o gerenciador de pacotes padrão do Python, o **pip**. Abra o seu terminal (talvez seja necessária permissão de administrador) e digite o comando **pip install requests**.

Obs: Por padrão, o **pip** faz uma instalação local da biblioteca, portanto, antes de executar este comando no terminal, esteja certo de estar na pasta certa do arquivo em que deseja usar este módulo. Para navegar entre pastas pelo terminal, você pode usar o comando **cd**, seguido do caminho para a pasta.

A instalação padrão do Thonny traz um interpretador próprio do Python consigo, no entanto, a instalação de bibliotecas externas via **pip install** é reconhecida apenas pelo interpretador padrão do seu sistema e não pelo da IDE. Para contornar este problema, você pode simplesmente alterar o interpretador padrão do Thonny clicando em **Tools > Options... > Interpreter**, e selecionar o interpretador Python, que deve ter como prefixo o local de instalação na sua máquina, veja abaixo:



A maioria das APIs que usamos são apenas para captação de dados, o modelo de requisição/resposta de uma API é, na verdade, muito simples. Ao acessar uma url padrão da API, o servidor responsável irá retornar um arquivo com dados que pode estar em diferentes formatos, – sendo os mais comuns o JSON e o XML –, ao invés de um website (arquivo .html), como estamos acostumados.

Logo, na forma mais básica de uso de uma API, precisamos apenas informar a url que queremos para o método **.get()** da biblioteca Requests, responsável por fazer a requisição ao servidor e retornar a resposta:

Existem várias APIs no mercado, algumas gratuitas, outras pagas, a maioria funciona através de uma chave de ativação, geralmente chamada apenas de “api key”. No entanto, há também algumas voltadas simplesmente para estudo, livres para acesso mesmo sem uma chave de ativação, como a que vamos usar nos nossos exemplos, a [Star Wars API](#).

```
apis.py
import requests

url = 'https://swapi.co/api'

result = requests.get(url)

print(result)

Shell

>>> %Run apis.py
<Response [200]>
```

O retorno do método **get()** é sempre uma resposta indicando o código de resposta do servidor para aquela url. Algumas destas respostas já devem ter aparecido para você em algum momento, como o **404**, que significa que a página não foi encontrada, ou o **500**, que indica um erro interno do servidor. O código **200**, por sua vez, indica que a requisição obteve resposta com sucesso.

Como mencionamos que a maioria das APIs exige o uso de uma “api key”, liberada normalmente mediante um cadastro prévio no website provedor desta, um código que pode aparecer com certa frequência é o **401**, que significa erro de autenticação.

Você pode acessar diretamente o código de status da resposta através do atributo **status_code** no retorno do método **.get()**:

```
apis.py
import requests

url = 'https://swapi.co/api'

result = requests.get(url)

print(result.status_code)

Shell

>>> %Run apis.py
200
```

Esse atributo te permite condicionar a sua tratativa com a resposta da requisição, afinal, você não quer tentar fazer as mesmas operações numa página não encontrada como numa requisição de sucesso.

```
if result.status_code == 200:
    print('Success!')
elif result.status_code == 404:
    print('Sorry, dude, I didn\'t find your url, please check it and try again')
```

Obviamente que nosso objetivo não é obter apenas o código de resposta da requisição, precisamos do conteúdo retornado. A biblioteca Requests possui três formas básicas para recuperar e tratar esse conteúdo, os atributos **.content**, **.text** e o método **.json()**.

O atributo de **.content** da resposta retorna o conteúdo “bruto” da requisição, em bytes:



```
apis.py
import requests

url = 'https://swapi.co/api'

result = requests.get(url)

print(result.content)
```

```
Shell
>>> %Run apis.py
b'{"people": "https://swapi.co/api/people/", "planets": "https://swapi.co/api/planets/", "films": "https://swapi.co/api/films/", "species": "https://swapi.co/api/species/", "vehicles": "https://swapi.co/api/vehicles/", "starships": "https://swapi.co/api/starships/"}'
```

O caracter “b” é como o Python representa um conteúdo em bytes.

O atributo **.text**, por sua vez, é um pouco diferente, retornando o conteúdo em formato de texto bruto, como uma **string**.



```
apis.py
import requests

url = 'https://swapi.co/api'

result = requests.get(url)

print(result.text)
```

```
Shell
>>> %Run apis.py
{"people": "https://swapi.co/api/people/", "planets": "https://swapi.co/api/planets/", "films": "https://swapi.co/api/films/", "species": "https://swapi.co/api/species/", "vehicles": "https://swapi.co/api/vehicles/", "starships": "https://swapi.co/api/starships/"}
```


Por fim, o método **.json()** transforma o conteúdo da resposta (se ele originalmente estiver em formato JSON) em um dicionário do Python.



```
apis.py
import requests

url = 'https://swapi.co/api'

result = requests.get(url)

dict = result.json()

print(type(dict))
print(dict)

Shell
>>> %Run apis.py

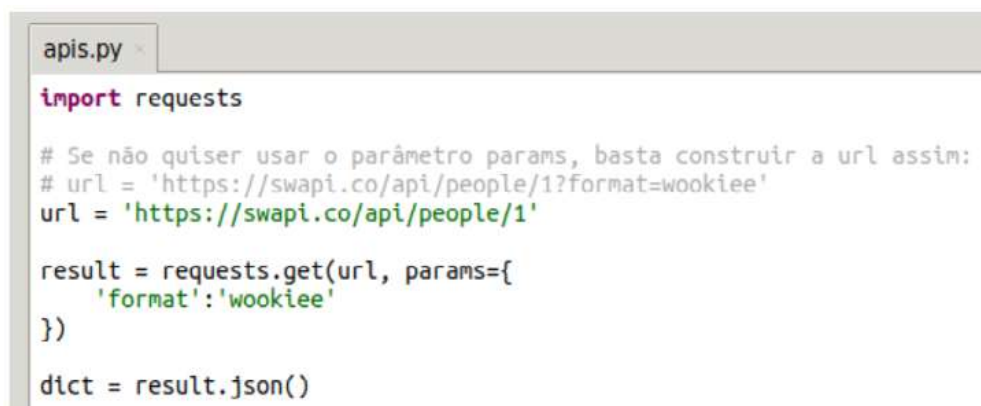
<class 'dict'>
{'people': 'https://swapi.co/api/people/', 'planets': 'https://swapi.co/api/planets/', 'films': 'https://swapi.co/api/films/', 'species': 'https://swapi.co/api/species/', 'vehicles': 'https://swapi.co/api/vehicles/', 'starships': 'https://swapi.co/api/starships/'}
```

Como nós já falamos, as APIs trabalham normalmente a partir de parâmetros que são passados na url. Os parâmetros via url podem ser passados na própria rota (entre barras) ou a partir de um símbolo de interrogação, nome do parâmetro e valor. Em caso de mais de um parâmetro, estes são separados por “&”. Exemplo:

<https://www.swapi.co/api/people/1?format=wookiee>

Neste exemplo, os dois tipos de passagem de parâmetros estão presentes, primeiro, informando o parâmetro people dessa url, com valor 1, e logo após, o parâmetro de formatação format, com valor wookiee (só use se souber a língua dos wookies).

Os parâmetros que aparecem a partir do símbolo de interrogação podem ser passados desta maneira diretamente na url, ou podem ser transmitidos através do query builder do método get(), veja como:



```
apis.py
import requests

# Se não quiser usar o parâmetro params, basta construir a url assim:
# url = 'https://swapi.co/api/people/1?format=wookiee'
url = 'https://swapi.co/api/people/1'

result = requests.get(url, params={
    'format': 'wookiee'
})

dict = result.json()
```

Apesar da biblioteca Requests fornecer recursos para nós tratarmos diferentes códigos de resposta, é importante ressaltar que mesmo um código de resposta de

erro só é retornado se for encontrado um servidor ativo responsável por aquela url ou domínio. Em outras palavras, se você construir uma url aleatória, só deixando seu gato passear pelo teclado ou batendo com a cabeça no computador., O Python apontará o erro, e quebrará o código.

11.3 Tratamento de exceções

Em vários momentos durante o nosso curso, nos deparamos com alguns cuidados que precisam ser tomados para evitar erros que podem acabar quebrando o nosso programa. Pois bem, o Python possui um recurso que nos permite obter (ou gerar) um erro e ainda assim impedir que o nosso algoritmo quebre. Estes erros são chamados de exceções e existe um bloco especial para tratá-las, que é o **try-except**:



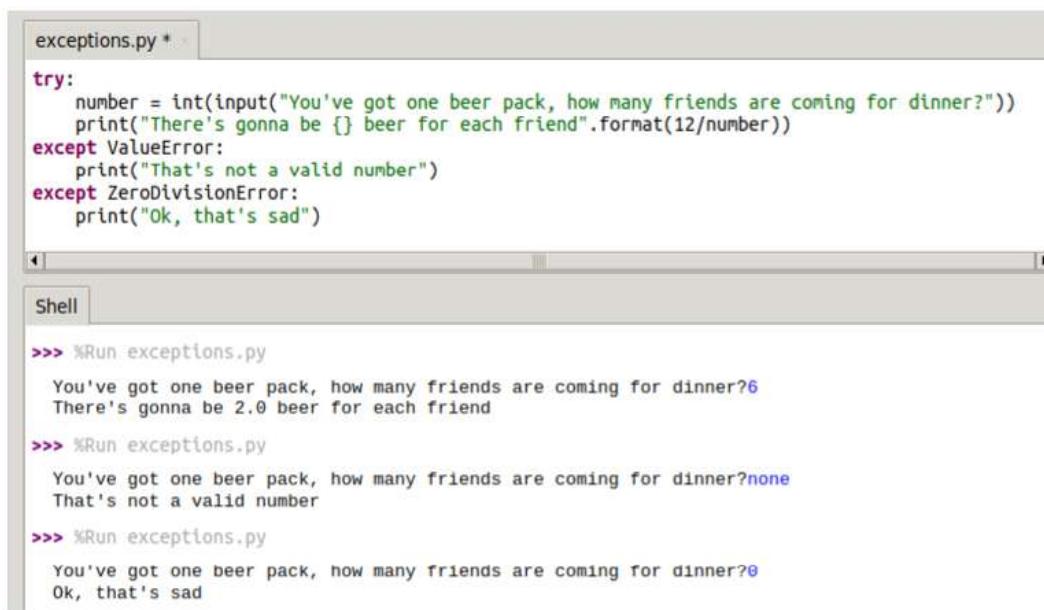
```
apis.py
import requests

try:
    url = 'https://swappi.co/api/people/1'
    result = requests.get(url, params={
        'format': 'wookiee'
    })
    print(result.json())
except Exception as err:
    print('Erro: ', err)

Shell
>>> %Run apis.py
Erro: HTTPSConnectionPool(host='swappi.co', port=443): Max retries exceeded with url: /api/people/1?format=wookiee (Caused by NewConnectionError('<urllib3.connection.VerifiedHTTPSConnection object at 0x7fcfe4a8deb8>: Failed to establish a new connection: [Errno -2] Name or service not known',))
```

Estes blocos funcionam da seguinte maneira, primeiro, os comandos listados após a cláusula **try** são executados, caso ocorra algum erro na execução de qualquer comando deste bloco, as demais linhas são ignoradas e a execução continua a partir do **except**. O bloco **except** pode ou não identificar o tipo de exceção que quer tratar, no exemplo acima, é identificada a classe genérica dos erros, a **Exception**.

É normal que tenhamos mais de um bloco **except** nessas tratativas quando diferentes erros são possíveis (parecido com o que fazemos com **if-elif**):



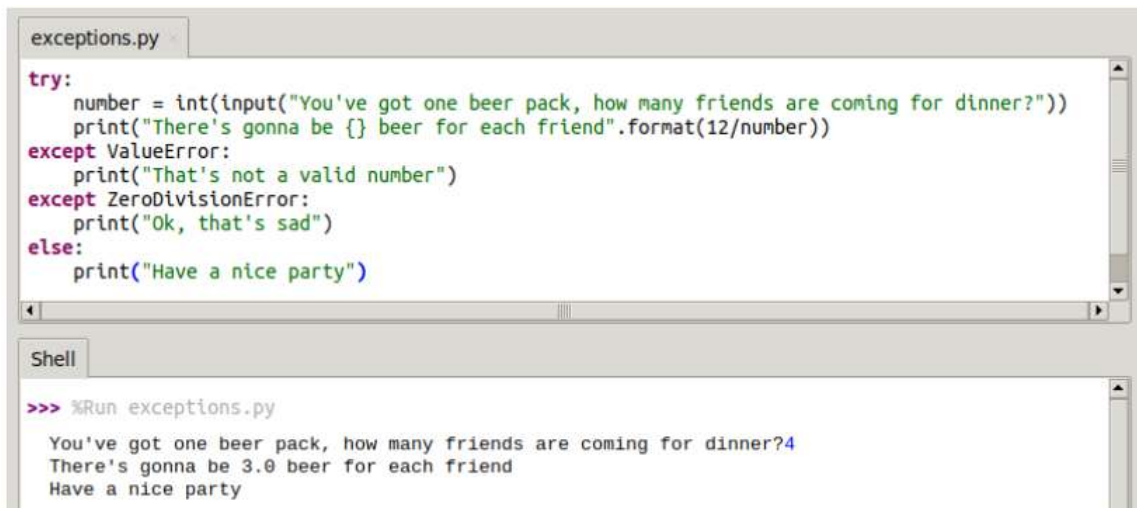
```
exceptions.py *
try:
    number = int(input("You've got one beer pack, how many friends are coming for dinner?"))
    print("There's gonna be {} beer for each friend".format(12/number))
except ValueError:
    print("That's not a valid number")
except ZeroDivisionError:
    print("Ok, that's sad")

Shell
>>> %Run exceptions.py
You've got one beer pack, how many friends are coming for dinner?6
There's gonna be 2.0 beer for each friend

>>> %Run exceptions.py
You've got one beer pack, how many friends are coming for dinner?none
That's not a valid number

>>> %Run exceptions.py
You've got one beer pack, how many friends are coming for dinner?0
Ok, that's sad
```

Normalmente, nós colocamos dentro do bloco **try** apenas os comandos que são passíveis de gerar erro, para as demais linhas de código, você pode abrigar dentro de um bloco **else**:



```
exceptions.py
try:
    number = int(input("You've got one beer pack, how many friends are coming for dinner?"))
    print("There's gonna be {} beer for each friend".format(12/number))
except ValueError:
    print("That's not a valid number")
except ZeroDivisionError:
    print("Ok, that's sad")
else:
    print("Have a nice party")

Shell
>>> %Run exceptions.py
You've got one beer pack, how many friends are coming for dinner?4
There's gonna be 3.0 beer for each friend
Have a nice party
```

Há ainda mais um bloco que pode ser utilizado nesse conjunto, para um comando que deve ser executado mesmo em casos de exceção, o bloco **finally**:



```
exceptions.py
try:
    number = int(input("You've got one beer pack, how many friends are coming for dinner?"))
    print("There's gonna be {} beer for each friend".format(12/number))
except ValueError:
    print("That's not a valid number")
except ZeroDivisionError:
    print("Ok, that's sad")
finally:
    print("Have a nice party")

Shell
>>> %Run exceptions.py
You've got one beer pack, how many friends are coming for dinner?4
There's gonna be 3.0 beer for each friend
Have a nice party

>>> %Run exceptions.py
You've got one beer pack, how many friends are coming for dinner?lots of
That's not a valid number
Have a nice party

>>> %Run exceptions.py
You've got one beer pack, how many friends are coming for dinner?0
Ok, that's sad
Have a nice party
```



CAPÍTULO

12

INTRODUÇÃO
A HTML E WEB
SCRAPING

12.1 Web Scraping

Embora haja um número enorme de APIs disponíveis no mercado, fornecendo dados dos mais variados setores, nem toda a informação disponível na web está disponível dentro de uma API, bem formatada em um arquivo JSON.

Algumas informações que você pode precisar estarão simplesmente como conteúdo de uma página web, seja dentro de um post de um blog, uma tabela, entre outros tantos exemplos.

Para saber como buscar esses dados, precisamos antes entender como uma página web é estruturada.

12.2 HTML

As páginas web são estruturadas através de uma linguagem de **formatação** chamada HTML (HyperText Markup Language). Esta linguagem de formatação é baseada numa muito próxima, chamada XML, há algum tempo atrás, houve até mesmo uma tentativa de fundir ambas as linguagens, dando origem ao XHTML. Embora o XHTML ainda seja reconhecido como

uma formatação e algumas páginas da web construídas se utilizando dele ainda existam, não se recomenda mais o seu uso. O órgão responsável por definir os seus protocolos de formatação tampouco continua oferecendo suporte para o seu uso.

As linguagens HTML e XML são baseadas em uma sintaxe de tags. As tags definem elementos e são delimitadas pelo sinal de menor que "<" e maior que ">". Podendo ou não vir em pares (tag de abertura e tag de fechamento). Tomemos como exemplo a tag que define tudo que compõe uma página html, a tag **html**.

```
<html>  
</html>
```

A maioria das tags são usadas em pares, aquelas que não possuem uma tag de abertura, são chamadas órfãs. Todo conteúdo entre as tags de abertura e fechamento de um elemento, pertencem a ele, inclusive outras tags. Por exemplo, observe a estrutura básica de um documento html:

```
<!DOCTYPE html>  
<html>  
  <head>  
    <title>Título da página</title>  
  </head>  
  <body>  
  
  </body>  
</html>
```

A primeira linha deste código define o tipo de documento que está sendo redigido. Neste caso, é especificado que o documento está sendo escrito na versão mais recente do HTML – versão 5 no momento em que esta apostila está sendo redigida. Como já discutido, as tags **html** compreendem tudo que pertencerá a definição da página, e possuem dois filhos principais, o par de tags **head** e o par de tags **body**.

Tudo o que é definido entre as tags **head** não estará visível na página, já que é um espaço para definição de codificação de caracteres, importação de arquivos externos e para o título da página, que você pode ver definido entre as tags **title**.

O conteúdo visível é o compreendido entre as tags **body**. Você encontra um guia das principais tags html no apêndice X desta apostila (**quick sheet de html).

As tags podem possuir atributos, os mais importantes para o processo de web Web Scraping são os chamados seletores. Existem dois seletores para identificação de tags **html**, o **id** e o **class**. O **id** é um identificador único, portanto deve haver apenas um elemento em cada página web com um determinado **id**, o seletor **class** funciona de maneira oposta, é um seletor utilizado para definir um padrão de elementos, agrupando vários elementos com um determinado atributo. Estes seletores não são exclusivos entre si, isto é, um elemento pode possuir simultaneamente **id** e **class**.

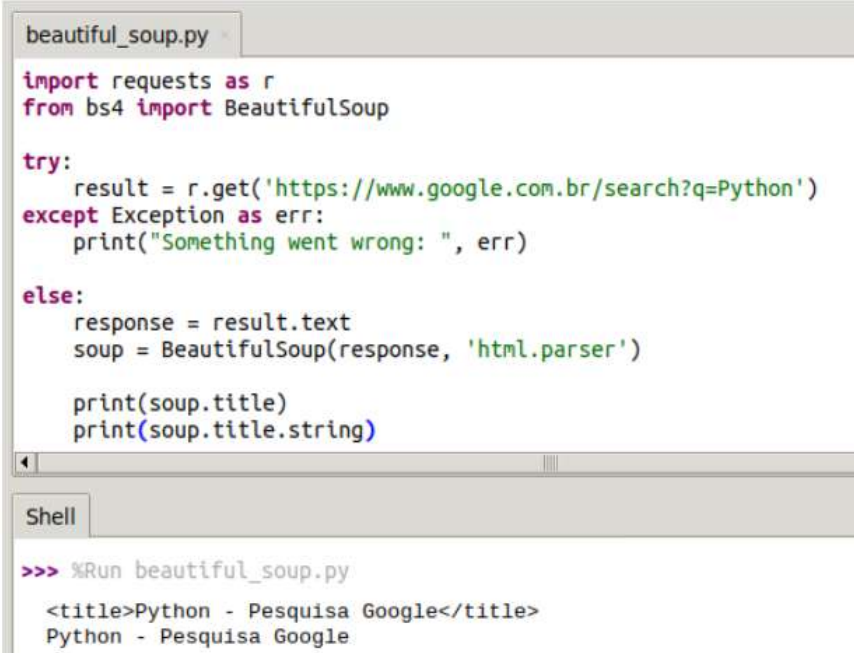
12.3 Beautiful Soup

A biblioteca **Beautiful Soup** é um módulo externo do Python que permite fácil navegação entre os elementos, seus conteúdos e atributos, facilitando a operação de Web Scraping, que é o nome dado a esta apuração de conteúdo

de uma página web a partir apenas da sua estruturação.

Como se trata de uma biblioteca externa, assim como o Requests, precisamos instalá-la manualmente via **pip**. Para instalar, basta abrir o terminal e executar o comando **pip install bs4**.

Da mesma maneira como fizemos para recuperar o conteúdo de uma página de uma API, podemos recuperar o conteúdo HTML de uma determinada página usando a biblioteca Requests:



```
beautiful_soup.py
import requests as r
from bs4 import BeautifulSoup

try:
    result = r.get('https://www.google.com.br/search?q=Python')
except Exception as err:
    print("Something went wrong: ", err)

else:
    response = result.text
    soup = BeautifulSoup(response, 'html.parser')

    print(soup.title)
    print(soup.title.string)
```

```
Shell
>>> %Run beautiful_soup.py
<title>Python - Pesquisa Google</title>
Python - Pesquisa Google
```

Para operar sobre o conteúdo **html** de uma página web usando a classe BeautifulSoup, basta instanciá-la, passando o conteúdo em texto para seu construtor e armazenando o retorno numa variável qualquer (no exemplo acima, **soup**). O retorno desta instância possui uma série de métodos e atributos que podem ser usados para recuperar dados do documento HTML.

Assim como fizemos para recuperar a tag de título da página do Google e seu conteúdo, podemos fazer com os demais elementos do documento. Vale lembrar que sempre que

tentarmos acessar uma tag diretamente pelo seu nome, será retornada a primeira tag encontrada com este nome.

Se quisermos acessar o valor do atributo de um elemento, basta informarmos o nome do atributo logo após, dentro de colchetes:

```
beautiful_soup.py
import requests as r
from bs4 import BeautifulSoup

try:
    result = r.get('https://www.google.com.br/search?q=Python')
except Exception as err:
    print("Something went wrong: ", err)

else:
    response = result.text
    soup = BeautifulSoup(response, 'html.parser')

    print(soup.p['class'])
```

Outra maneira de recuperar diretamente um elemento pelo seu nome é usando o método **find()**, que aceita o nome do atributo ou um atributo com determinado valor (sendo que o atributo **class** é escrito como **class_** para não violar o uso de palavras de reservadas), veja abaixo:

```
beautiful_soup.py
import requests as r
from bs4 import BeautifulSoup

try:
    result = r.get('https://docs.python.org/3/')
except Exception as err:
    print("Something went wrong: ", err)

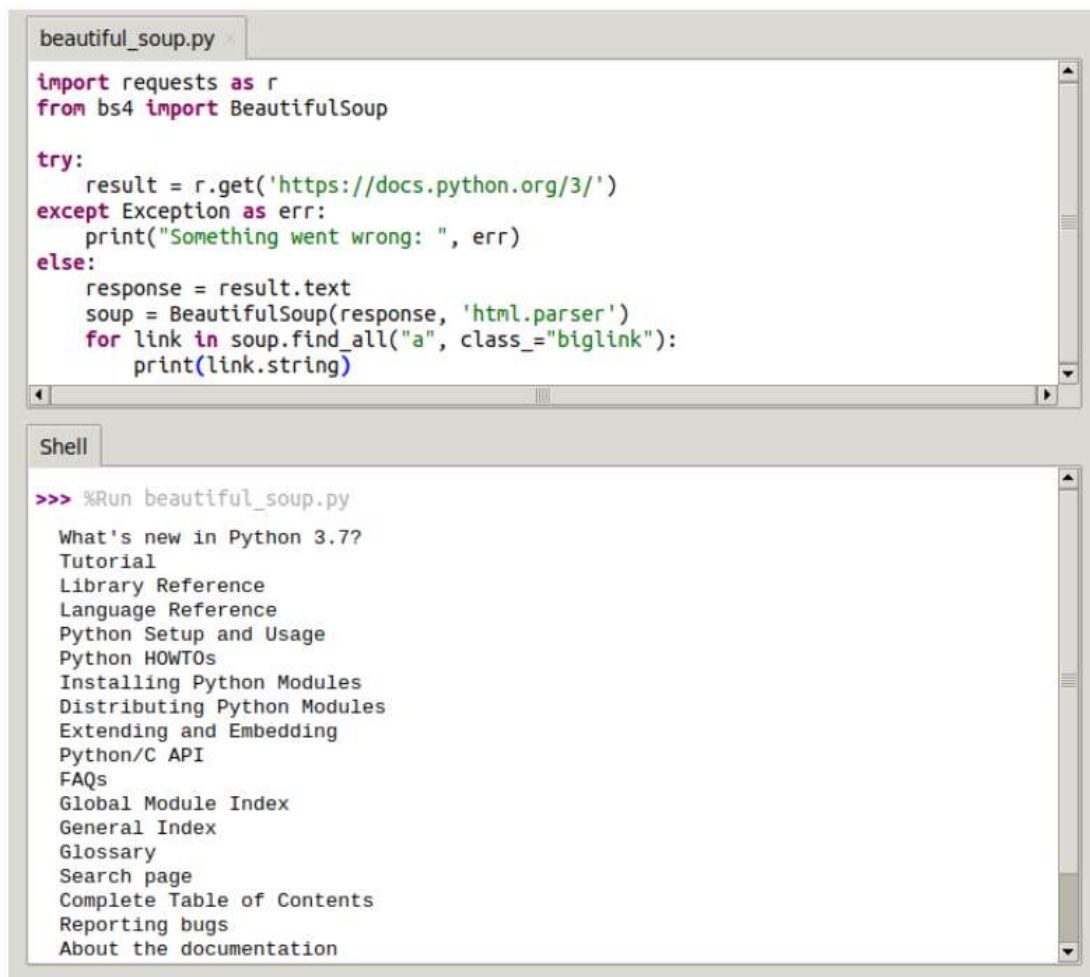
else:
    response = result.text
    soup = BeautifulSoup(response, 'html.parser')

    print(soup.find("a", class_="biglink").prettify())
```

```
Shell
>>> %Run beautiful_soup.py
<a class="biglink" href="whatsnew/3.7.html">
  What's new in Python 3.7?
</a>
```

O método **prettify()**, no exemplo acima, é utilizado para “organizar” o conteúdo textual retornado, corrigindo a indentação das tags e seus conteúdos.

É normal, no entanto, que queiramos operar sobre vários elementos com um mesmo atributo ou um mesmo tipo, para isso, basta usarmos o método **find_all()**, que aceita os mesmos parâmetros que o método **find()**, mas retorna todos os elementos encontrados em uma lista.

The image shows a Python IDE window with two panes. The top pane, titled 'beautiful_soup.py', contains a Python script that uses the requests and BeautifulSoup libraries to fetch the Python 3.7 documentation page and print out all links with the class 'biglink'. The bottom pane, titled 'Shell', shows the output of running the script, which lists various documentation links like 'What's new in Python 3.7?', 'Tutorial', 'Library Reference', etc.

```
beautiful_soup.py
import requests as r
from bs4 import BeautifulSoup

try:
    result = r.get('https://docs.python.org/3/')
except Exception as err:
    print("Something went wrong: ", err)
else:
    response = result.text
    soup = BeautifulSoup(response, 'html.parser')
    for link in soup.find_all("a", class_="biglink"):
        print(link.string)
```

```
Shell
>>> %Run beautiful_soup.py

What's new in Python 3.7?
Tutorial
Library Reference
Language Reference
Python Setup and Usage
Python HOWTOs
Installing Python Modules
Distributing Python Modules
Extending and Embedding
Python/C API
FAQs
Global Module Index
General Index
Glossary
Search page
Complete Table of Contents
Reporting bugs
About the documentation
```


Textos

Leonardo Paz
Luís Gustavo Sales
Pietro Ribeiro

Design

Ricardo Souza

Vetores

freepik.com

Propriedade intelectual da Let's Code Academy ©

[illegible]