Básico de Python

Sergio Pedro Rodrigues Oliveira

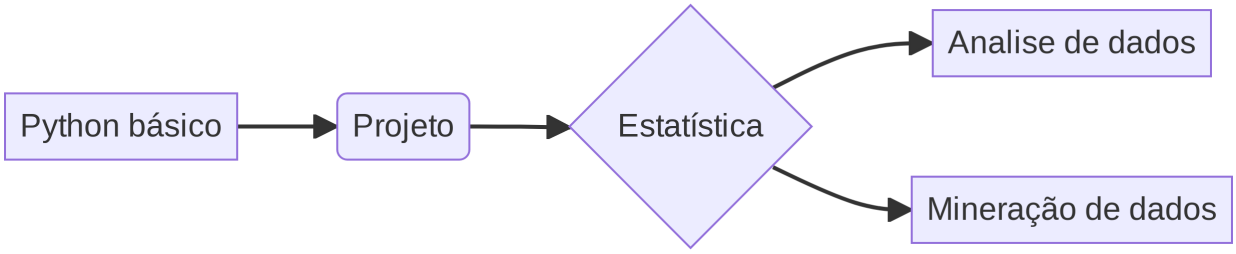
05 February 2025

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

1 Diagrama de estudo



2 Variáveis e tipos de dados simples

2.1 print

2.1.1 `print()`

Print é uma função que exibe uma string na tela.

Exemplo:

```
print("string")
```

```
string
```

2.1.2 `print()` com variáveis

Podemos usar a função `print()` para imprimir uma variável string.

Exemplo:

```
message = "Hello world!"
print(message)
```

```
Hello world!
```

2.1.3 `.format()`

O método `.format()` se associa a função `print`, para ajudar na formatação. Inserimos chaves `{}` na string e dentro do `.format()` inserimos as variáveis conforme as chaves aparecem no texto.
Exemplo:

```
print("My name is {fname}, I'm {age}".format(fname = "John", age = 36))
```

```
My name is John, I'm 36
```

Também dentro das chaves podemos inserir os elementos de formação, como podem ser observados na Table 1.
Exemplo:

```
print("My name is {fname}, I'm {age:.2f}".format(fname = "John", age = 36))
```

```
My name is John, I'm 36.00
```

2.1.4 f-string - *formatted string literals*

Em um programa Python, as f-strings são iniciadas com a letra f ou F, contendo expressões envolvidas por um par de chaves `{...}`, modificadas dentro da string a ser formatada. As f-strings consideram tudo que está fora do par de chaves como sendo texto literal e, portanto, na saída, o texto será replicado sem nenhuma alteração.
A maneira mais simples de formatar uma string é informando os valores que a irão compor.
Exemplo:

```
#Importando biblioteca e função
from datetime import datetime

#Variáveis
ano_atual = datetime.now().year
clube = "CRF"
campeonato_mundial = 1
ano_fundacao = 1895

#f-string
print(f"{clube} possui {campeonato_mundial:02d} título mundial.")
print(f"São {ano_atual - ano_fundacao:.2f} anos de existência.")
```

```
CRF possui 01 título mundial.
São 130.00 anos de existência.
```

São atribuidos valores as variáveis `clube`, `campeonato_mundial`, `ano_atual` e `ano_fundacao`, respectivamente. As posições em que se encontram as expressões serão substituidas pelos seus respectivos valores. A expressão `{ano_atual - ano_fundacao:.2f}` será calculada.

Podemos concatenar linhas adicionando *f* na frente da *string*.
Exemplo:

```
#Importando biblioteca e função
from datetime import datetime

#Variáveis
ano_atual = datetime.now().year
clube = "CRF"
campeonato_mundial = 1
ano_fundacao = 1895

#f-string
print(f"O {clube} foi fundado em 17 de novembro de {ano_fundacao}. "
f"\nEm seus {ano_atual - ano_fundacao:.2f} anos de história, "
f"obteve {campeonato_mundial:02d} título mundial.")
```

```
O CRF foi fundado em 17 de novembro de 1895.
Em seus 130.00 anos de história, obteve 01 título mundial.
```

A expressão `:02d` e `:.2f` dentro das chaves formatam os valores para respectivamente: um número inteiro com 2 digitos, se ele possuir menos de 2 digitos, preencha com zeros a esquerda; e que deseja trabalhar com ponto flutuante e a quantidade de casas decimais. Os tipos de formatação podem ser obsevados na Table 1.

Table 1: Formatações

Símbolo	Formatação
:<	Alinha o resultado à esquerda (dentro do espaço disponível).
:>	Alinha o resultado à direita (dentro do espaço disponível).
:^	Alinha o resultado centralizado (dentro do espaço disponível).
:=	Coloca o sinal na posição mais à esquerda.
:+	Usa um sinal de mais para indicar se o resultado é positivo ou negativo.
:-	Usa um sinal de menos apenas para valores negativos.
:	Usa um espaço para inserir um espaço extra antes dos números positivos (e um sinal de menos antes dos números negativos).
:,	Usa uma vírgula como separador de milhar.
:_	Usa um underscore como separador de milhar.
:b	Formato binário.
:c	Converte o valor no caractere Unicode correspondente.
:d	Formato decimal.
:e	Formato científico, com e minúsculo.
:E	Formato científico, com E maiúsculo.
:f	Converte para formato de ponto flutuante.
:F	Converte para formato de ponto flutuante, em formato maiúsculo (mostrar inf e nan como INF e NAN).
:g	Formato geral.
:G	Formato geral (usando E maiúsculo para notações científicas).
:o	Formato de octal.
:x	Formato hexadecimal, letras minúsculas.
:X	Formato hexadecimal, letras maiúsculas.
:n	Formato de número.
:%	Formato de porcentagem.

2.2 Regras de nomes de variáveis

Regras ou diretrizes para usar variáveis em Python.

- Nomes de variáveis deve conter apenas letras, números e underscores. Podemos começar a variável com letra ou underscore, mas nunca com um número.
- Espaços não são permitidos em nomes de variáveis, mas underscores podem ser usados para separar palavras.
- Evite usar palavras reservadas e nome de funções em Python como nome de variáveis.
- Nomes de variáveis devem ser concisos, porém descritivos.
- Tome cuidado ao usar a letra l e a letra maiuscula O, pois podem ser confundidas com os números 1 e 0.

2.3 Concatenando strings

Podemos usar o simbolo de (+) para combinar strings (concatenar).

Exemplo:

```
first_name = "ada"
last_name = "lovelace"
full_name = first_name + " " + last_name
print("Hello, " + full_name.title() + "!")
```

```
Hello, Ada Lovelace!
```

Os espaços em branco entre aspas servem para criar espaços na string.

2.4 Métodos auxiliares da função `print()`

1. `.title()`

Coloca apenas as primeiras letras em maiúsculas de cada palavra e o resto em minúscula.

Exemplo:

```
full_name = "ada lovelace"
print(full_name.title())
```

```
Ada Lovelace
```

2. `.upper()`

Coloca todas as letras em maiúsculas.

Exemplo:

```
full_name = "ada lovelace"
print(full_name.upper())
```

```
ADA LOVELACE
```

3. `.lower()`

Coloca todas as letras em minusculas. O método `.lower()` é particularmente útil para armazenar dados. Converter os dados em minúscula antes de armazenar.

Exemplo:

```
full_name = "ada lovelace"
print(full_name.lower())
```

```
ada lovelace
```

2.5 Caracteres de escape

Podemos inserir alguns caracteres de escape no texto para executar alguma ação, como pular linha, gerar tabulação e etc. Alguns caracteres podem ser vistos na Table 2.

Todos os caracteres de escape começam com barra(\) + complemento.

Table 2: Caracteres de escape

Caracteres de escape	Descrição
\t	Gera tabulação (tab).
\n	Gera quebra de linha.

Exemplo:

```
print("Language:\nPython\nJava\nC\nJavaScript")
```

```
Language:
Python
Java
C
JavaScript
```

2.6 Removendo espaços em branco `print()`

1. `.rstrip()`

Remove espaço em branco do lado direito.

Exemplo:

```
favorite_linguage = 'python '
favorite_linguage.rstrip()
```

```
'python'
```

2. `.lstrip()`

Remove espaço em branco do lado esquerdo.

Exemplo:

```
favorite_linguage = ' python'
favorite_linguage.lstrip()
```

```
'python'
```

3. `.strip()`

Remove os espaços em branco dos dois lados ao mesmo tempo.

Exemplo:

```
favorite_linguage = ' python '
favorite_linguage.strip()
```

```
'python'
```

- Os metodos usados não removem os espaços em branco em definitivo, para remover em definitivo é necessario armazenar o valor novo na variável.

```
favorite_linguage = ' python '
favorite_linguage = favorite_linguage.strip()
favorite_linguage
```

```
'python'
```

2.7 Números

A linguagem Python faz tipagem automática (dinâmica), tipa a variável de acordo com o uso. E o Python contém uma tipagem forte, não faz converção automática do tipo de uma variável para executar uma ação (operação).

Em resumo, python tem é uma linguagem de tipagem dinâmica e forte.

Os principais tipos de dados no Python são estão presentes na Table 3.

Table 3: Principais tipos de dados

Nome	Abreviação	Descrição
Inteiro	int	Números inteiros
Ponto flutuante	float	Números com ponto decimal

2.8 Funções de conversão de tipo

Podemos converte variáveis para determinado tipo especificado usando funções de conversão de tipo, como pode ser obeservado na Table 4.

Converter uma variável não é permanente, a não ser que a ação seja armazenada na variável explicitamente.

Table 4: Funções de conversão de tipo

Tipo para converter	Função	Descrição
int	`int()`	Converte variável para o tipo inteiro(int)
float	`float()`	Converte variável para o tipo float
string	`str()`	Converte variável para o tipo string

A função `str()` é deveras importante, pois pode auxiliar na função `print()`. A função `print()` só imprime na tela veriáveis string, sendo assim, precisamos converter as variáveis de outros tipos para string (pelo menos, momentaneamente), para comprir essa condição.

Exemplo:

```
age = 23
print("Happy " + str(age) + "rd Birthday!")
```

```
Happy 23rd Birthday!
```

2.9 Descobrindo o tipo da variável usando a função `type()`

Podemos usar a função `type()` para descobrir o tipo de determinada variável.

```
age = 23
print (type(age))
```

```
<class 'int'>
```

É uma **boa pratica** usar a função `type()`, para conferir o tipo da variável, antes de manipular alguma variável. Assim o programador terá o controle sobre as variáveis que esta trabalhando. Essa boa prática evita erros.

Também é uma **boa prática**, ao identificar/observar um erro, conferir os tipos das variáveis envolvidas. É um dos erros mais comuns: erro de tipagem.

2.10 Operações básicas

A Table 5 apresenta as principais operações básicas do python.

Table 5: Operações básicas

Operação	Símbolo	Exemplo
Soma	+	2+2=4
Subtração	-	3-2=1
Multiplicação	*	2*3=6
Divisão	/	5/4=1.25
Divisão inteira	//	5//4=1
Resto da divisão (módulo)	%	10%8=2
Potência	**	3**2=9
Raiz	**	4**0.5=2

2.11 Biblioteca `math` para ampliar operações matematicas

Podemos usar o pacote `math` para ampliar as funções matemáticas do Python (básicas, trigonométricas e estatísticas). A Table 6 apresenta as principais funções básicas da biblioteca `math`.

Table 6: Algumas operações da biblioteca `math`

Operação	Símbolo	Exemplo
Soma	`math.add(x,y)`	`math.add(2,2) = (2+2)=4`
Subtração	`math.subtract(x,y)`	`math.subtract(2,2) = (2-2)=0`
Raiz quadrada	`math.sqrt()`	`math.sqrt(4)=2`
Potência	`math.pow(x,y)`	`math.pow(2,3) = (2**3)=8`
Seno	`math.sin()`	`math.sin()`, retorna um ângulo em radianos.
Cosseno	`math.cos()`	`math.cos()`, retorna um ângulo em radianos.
Tangente	`math.tan()`	`math.tan()`, retorna um ângulo em radianos.
potencia de Euler	`math.exp(x)`	`math.exp(x) = math.pow(math.e**x)`
Logaritmo natural, ou log neperiano	`math.log(x)`	`math.log(2)=0.69`
Logaritmo	`math.log(x[,base])`	`math.log(2,10)=0.3`

Para converter o ângulo para radianos podemos usar a função `math.radians()`.

```
import math
#Seno do ângulo de 45°
#Resultado em Radianos
print(str(math.sin(math.radians(45))))
```

```
0.7071067811865475
```

Para converter de radiano para grau podemos usar a função `math.degrees()`.

```
import math
#Seno do ângulo de 45°
#Resultado em ângulo
print(str(math.degrees(math.sin(math.radians(45)))))
```

```
40.51423422706977
```

2.12 Operações lógicas básicas

A Table 7 apresenta as principais operações lógica básica do python. As operações lógicas retornam `True` ou `False`.

Table 7: Operações Lógicas Básicas

Operação	Nome	Função	Exemplo
==	Igual a	Varifica se um valor é igual ao outro.	1==1 = True
!=	Diferente de	Varifica se um valor é diferente ao outro.	1!=2 = True
>	Maior que	Varifica se um valor é maior que outro.	5>1 = True
>=	Maior ou igual	Varifica se um valor é maior ou igual a outro.	5>=5 = True
<	Menor que	Varifica se um valor é menor que outro.	1<5 = True
<=	Menor ou igual	Varifica se um valor é menor ou igual a outro.	1<=4 = True
and	E	Retorna True se ambas as afirmações forem verdadeiras.	(1==1) and (4<5)
or	Ou	Retorna True se uma das afirmações for verdadeiras.	(1==1) or (2<1) =
not	Negação	Retorna Falso se o resultado for verdadeiro, ou o contrario.	not (1==1) = Fals

2.13 Operadores de identidade

Os operadores de identidade, Table 8, são utilizados para comparar objetos, se os objetos testados referenciam o mesmo objeto.

Table 8: Operadores identidade

Operador	Definição
is	Retorna `True` se ambas as variáveis são o mesmo objeto.
is not	Retorna `True` se ambas as variáveis não são o mesmo objeto.

Exemplo de operações de identidade:

```
lista = [1,2,3]
outra_lista = [1,2,3]
recebe_lista = lista

print(f"São o mesmo objeto: {lista is outra_lista}")
```

```
São o mesmo objeto: False
```

```
lista = [1,2,3]
outra_lista = [1,2,3]
recebe_lista = lista

print(f"São o mesmo objeto: {lista is recebe_lista}")
```

```
São o mesmo objeto: True
```

2.14 Operações de associação

Os operadores de associação, Table 9, servem para verificar se determinado objeto esta **associado** ou **pertence** a determinada estrutura de dados.

Table 9: Operadores de associação

Operação	Função
in	Retorna `True` caso valor seja encontrado na sequência.
not in	Retorna `True` caso valor não seja encontrado na sequência.

Exemplos de operações de associação:

```
lista = ["Python", 'Academy', "Operadores", 'Condições']
print('Python' in lista)
```

```
True
```

```
lista = ["Python", 'Academy', "Operadores", 'Condições']
print('SQL' not in lista)
```

```
True
```

2.15 Comentários

Um comentário permite escrever notas em seus programas em liguagem natural. Em Python, o caractere sustenido (#) indica um comentário. Tudo que vier depois de um caractere sustenido en seu código será ignorado pelo interpretador Python.

Boas práticas em comentários:

1. Explicar o que o código deve fazer.
2. Como faz para funcionar.

2.16 Zen Python

É um guia de **boas práticas**.

```
import this
```

```
The Zen of Python, by Tim Peters

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

Principais pontos:

1. Bonito é melhor do que feio.
2. Simples é melhor que complexo.
3. Complexo é melhor que complicado.
4. Legibilidade conta.
5. Deve haver uma - e, de preferência, apenas uma - maneira óbvia de fazer algo.
6. Agora é melhor que nunca.

3 Listas

3.1 Lista

Uma lista é uma coleção de itens em uma ordem em particular. Os colchetes([]) indicam uma lista e os elementos individuais de uma lista são separados por vírgula. [ver 1, p. 71]

Exemplo:

```
bicycles = ['trek','cannondale','redline','specialized']
print(bicycles)
```

```
['trek', 'cannondale', 'redline', 'specialized']
```

3.2 Acessando elementos de uma lista

Podemos acessar a qualquer item de uma lista informando a posição, ou índice. As posições de uma lista começam no 0, e não no 1.

Para acessar um elemento de uma lista, informamos o nome nome da lista seguido do índice do item entre colchetes.

Exemplo:

```
#Acessando o primeiro item da lista
bicycles = ['trek','cannondale','redline','specialized']
print(bicycles[0].title())
```

```
Trek
```

Para acessar a lista de trás pra frente podemos usar a posição invertida seguida do símbolo de menos na frente. Sendo assim, a posição do último item é -1, do penúltimo -2 e assim sucessivamente.

Exemplo:

```
#Acessando o último item da lista
bicycles = ['trek','cannondale','redline','specialized']
print(bicycles[-1].title())
```

```
Specialized
```

3.3 Alterando, acrescentando e removendo elementos

Dado que a lista é um elemento dinâmico (pode, e provavelmente ocorrerá, de sofrer modificações com o uso), este tópico comentará os principais formas de modificação de listas.

3.3.1 Modificando elementos de uma lista

Para alterar um elemento que você quer modificar, use o nome da lista seguido do índice do elemento que quer modificar, e então forneça um novo valor.

```
#Alterando o item 1 da lista (índice 0)
motorcycles = ['honda','yamaha','suzuki']
motorcycles[0] = 'ducati'
print(motorcycles)
```

```
['ducati', 'yamaha', 'suzuki']
```

3.3.2 Acrescentando elementos em uma lista

Existem diversas formas de adicionar elementos a uma lista:

3.3.2.1 Concatenando elementos no final de uma lista, método `.append()`

Adiciona um novo elemento no final da lista usando o método `.append()`.

Exemplo:

```
#Adicionando elemento ao final da lista
motorcycles = ['honda','yamaha','suzuki']
motorcycles.append('ducati')
print(motorcycles)
```

```
['honda', 'yamaha', 'suzuki', 'ducati']
```

3.3.2.2 Inserindo elementos em uma lista, método `.insert()`

Este método insere um elemento em determinada posição da lista, usando o método `.insert(índice,elemento)`.

Exemplo:

```
#Adicionando um item na segunda posição da lista (índice 1)
motorcycles = ['honda','yamaha','suzuki']
motorcycles.insert(1,'ducati')
print(motorcycles)
```

```
['honda', 'ducati', 'yamaha', 'suzuki']
```

3.3.3 Removendo elementos de uma lista

Os métodos para remover um item, ou um conjunto de itens, de uma lista.

3.3.3.1 Instrução `del`

Se a posição do item que você quer remover de uma lista for conhecida, a instrução `del` remove (deleta) um item em qualquer determinada posição. Depois de removido (deletado) não podemos mais acessar o valor, quando usado a instrução `del`.

```
# Remover (deletar) primeiro item da lista, indíce 0
motorcycles = ['honda','yamaha','suzuki']
print(motorcycles)

del motorcycles[0]
print(motorcycles)
```

```
['honda', 'yamaha', 'suzuki']
['yamaha', 'suzuki']
```

3.3.3.2 Método `.pop()`

Existem duas formas de usar o método `.pop()`:

1. `.pop()`

As vezes há necessidade de usar o valor de um item depois de removê-lo de uma lista. O método `.pop()` remove o **último** item de uma lista, mas permite que você trabalhe com esse item depois da remoção.

Remove o primeiro item de uma pilha, ou seja, o último item de uma lista.

Para usarmos o item removido é necessário, salva-lo numa variável.

Exemplo:

```
# Uso do método .pop()
# Removendo último item da lista e
# Trabalhando com o item removido.
motorcycles = ['honda','yamaha','suzuki']
print(motorcycles)

pop_motorcycle = motorcycles.pop()
```

```
print(motorcycles)
print(pop_motorcycle)
```

```
['honda', 'yamaha', 'suzuki']
['honda', 'yamaha']
suzuki
```

2. `.pop(índice)`

Podemos usar o `.pop()` para remover um item em qualquer posição em uma lista, se incluirmos o índice do item que você deseja remove entre parênteses.

Exemplo:

```
# Uso do método .pop()
# Removendo o segundo item da lista e
# Trabalhando com o item removido.
motorcycles = ['honda','yamaha','suzuki']
print(motorcycles)

pop_motorcycle = motorcycles.pop(1)
print(motorcycles)
print(pop_motorcycle)
```

```
['honda', 'yamaha', 'suzuki']
['honda', 'suzuki']
yamaha
```

3.3.3.3 Método `.remove()`

Remove um item de acordo com o valor. É usado quando sabemos o valor do item, mas não a posição.

O método `.remove()` apaga apenas a primeira ocorrência do valor especificado. Para apagar mais de uma ocorrência será necessario o uso de um laço, para cada ocorrência.

Exemplo:

```
# Uso do método .remove()
# Removendo um item da lista pelo valor
motorcycles = ['honda','yamaha','ducati']
print(motorcycles)
```

```
too_expensive = 'ducati'
motorcycles.remove(too_expensive)
print(motorcycles)
```

```
['honda', 'yamaha', 'ducati']
['honda', 'yamaha']
```

3.4 Organizando uma lista

Dado que com frequência, as listas são organizadas numa ordem imprevisível, se torna necessario organizar as informações em uma ordem particular. O Python tem mecanismos para organizar listas. São eles:

3.4.1 Método `.sort()`

Ordena uma lista em ordem alfabética, ou alfabetica inversa.

Para ordenar uma lista em ordem alfabética inversa, basta passar o argumento `reverse = True` para o método `.sort()`.

Uma vez ordenada pelo método `.sort()` a lista não retorna a ordem original (ordenação permanente).

Exemplo:

```
# Ordenando a lista cars usando o método .sort()
cars = ['bmw','audi','toyota','subaru']
print(cars)
cars.sort()
print(cars)
cars.sort(reverse=True)
print(cars)
```

```
['bmw', 'audi', 'toyota', 'subaru']
['audi', 'bmw', 'subaru', 'toyota']
['toyota', 'subaru', 'bmw', 'audi']
```

3.4.2 A função `sorted()`

A função `sorted()` ordena uma lista de forma temporaria, não altera a lista original, em ordem alfabetica. Ou seja, a lista volta a forma orginal ao final do uso da função.

Assim como no médodo `.sort()`, podemos ordenar a lista em ordem alfabética inversa adicionando o argumento `reverse=True`.

Exemplo:

```
# Ordenando temporariamente a lista cars usando a função sorted()
cars = ['bmw','audi','toyota','subaru']
print(cars)
print(sorted(cars))
print(sorted(cars,reverse=True))
print(cars)
```

```
['bmw', 'audi', 'toyota', 'subaru']
['audi', 'bmw', 'subaru', 'toyota']
['toyota', 'subaru', 'bmw', 'audi']
['bmw', 'audi', 'toyota', 'subaru']
```

3.4.3 Método `.reverse()`

Para inverter a ordem original de uma lista, podemos usar o método `.reverse()`.

O método `.reverse()` não organiza a lista em ordem alfabética inversa, o método inverte a lista original.

O método `.reverse()` ordena de forma permanente a lista, porém se usarmos o método novamente, teremos a lista original. Logo, é fácil reverter o uso do método `.reverse()`.

Exemplo:

```
# Método .reverse() para inverte, de modo permanete, a ordem da lista.
cars = ['bmw','audi','toyota','subaru']
print(cars)
cars.reverse()
print(cars)
cars.reverse()
print(cars)
```

```
['bmw', 'audi', 'toyota', 'subaru']
['subaru', 'toyota', 'audi', 'bmw']
['bmw', 'audi', 'toyota', 'subaru']
```

3.5 Descobrindo o tamanho de uma lista - `len()`

Podemos descobrir o tamanho de uma lista usando a função `len()`.

Exemplo:

```
cars = ['bmw','audi','toyota','subaru']
len(cars)
```

```
4
```

3.6 Contando determinado elementos - `.count()`

Conta quantas ocorrência de um determinado elemento existe em uma lista.
`ocorrencias = lista.count("elemento")`

Exemplo:

```
atrizes = ["adriana", "adriana", "camila",
           "danielle", "fernanda", "helena",
           "paolla", "raquel", "viola"]
ocorrencias = atrizes.count("adriana")
print(f"Número de ocorrências da adriana: {ocorrencias}")
```

```
Número de ocorrências da adriana: 2
```

3.7 Índice - `.index()`

É utilizado para retornar o índice em que se encontra a primeira ocorrência de um elemento informado.
`lista.index("elemento")`

Exemplo:

```
atrizes = ["adriana", "adriana", "camila",
           "danielle", "fernanda", "helena",
           "paolla", "raquel", "viola"]
indice = atrizes.index("raquel")
print(f"Índice na lista da raquel: {indice}")
```

```
Índice na lista da raquel: 7
```

4 Trabalhando com listas

4.1 Percorrendo uma lista inteira com um laço

Podemos usar um laço `for` para percorrer toda uma lista, podendo assim entre outras coisas, efetuar tarefas em cada item da lista.

A estutura básica do `for` é:

```
for variável_nova in lista :
  tarefas
```

O laço diz para a cada iteração pegar um elemento da lista e armazenar na nova variável, e executar uma tarefa a cada iteração. Toda tarefa indentada depois dos dois pontos é considerada dentro do laço.

No Python o `for`, usa indentação para determinar o que esta dentro do laço.

Qual quer linha após o laço que não for indentada é considerada fora do laço.

Exemplo:

```
#Executando um laço com base numa lista
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
  print(magician)
```

```
alice
david
carolina
```

4.2 Erros comuns de indentação

- Esquecer de indentar.
- Esquecer de indentar linhas adicionais do laço.
- Indentação desnecessaria.
- Indentando desnecessariamente após o laço.
- Esquecer os dois-pontos do laço `for`.

4.3 Listas numéricas

4.3.1 Gerando série de números com a função `range()`

A função `range()` é usada para gerar uma série de números, de uma determinada sequência numérica.

A função `range()` faz o Python começar a contar no primeiro valor definido (limite inferior) e parar quando atingir o segundo valor definido (limite superior). Como o for para no segundo valor, a saída não conterá o valor final. Também podemos definir um intervalo, pulando alguns valores.

Estrutura da função `range()`:

```
range(limite_inferior, limite_superior, intervalo)
```

Exemplo:

```
for value in range(1,5):
  print(value)
```

```
1
2
3
4
```

4.3.2 Usando `range()` para gerar uma lista - `list()`

Podemos usar para criar uma lista de números, combinando a função `range()`, que gera uma série númerica, com a função `list()`, que cria um lista.

Exemplo:

```
numbers = list(range(1,6))
print(numbers)
```

```
[1, 2, 3, 4, 5]
```

Exemplo 2:

```
numbers = list(range(2,11,2))
print(numbers)
```

```
[2, 4, 6, 8, 10]
```

4.3.3 Estatística simples com lista de números

As principais funções estatísticas estão contidas na Table 10.

Table 10: Estatística simples

Funções	Descrição
`min()`	Retorna o valor mínimo.
`max()`	Retorna o valor máximo.
`sum()`	Somatório.

As principais bibliotecas auxiliares de funções estatísticas são:

1. `math`
2. `numpy` as np
3. `statistics`
4. `Pandas` as pd

Medidas de posição utilizando bibliotecas python, Table 11.

Table 11: Medidas de posição, bibliotecas python

Funções	Descrição
`np.mean()`	Média aritmética
`statistics.median()`	Mediana
`statistics.mode()`	Moda
`np.quantiles(array, 0.5)`	Quartil
`np.percentile(array, 50)`	Percentil

Medidas de dispersão utilizando bibliotecas python, Table 12.

Table 12: Medidas de dispersão, bibliotecas python

Funções	Descrição
`pd.var()`	Variância
`pd.std()`	Desvio-padrão
`pd.mad()`	Desvio absoluto
`pd.cov()`	Covariância
`pd.corr()`	Correlação

4.4 list comprehensions

List comprehensions é uma forma de criar listas já acoplando o laço for nelas, deixando o código mais enxuto.

Sintaxe:

```
nome_lista = [expressão_calculada_do_for for variável in range()]
```

Exemplo:

```
squares = [value ** 2 for value in range(1,11)]
print(squares)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

4.5 Trabalhando com parte de uma lista

Neste tópico vamos trabalhar com um grupo de itens de uma lista, no Python é chamado de *fatia* (de uma lista).

4.5.1 Fatiando uma lista

1. Fatia simples

Para criar uma fatia, especifique o índice do primeiro e o último elemento com os quais você deseja trabalhar.

O Python para em um item antes do segundo índice (índice final) especificado.

Exemplo:

```
#Exibindo os 3 primeiros elementos de uma lista.
players = ["charles","martina","michael","florence","eli"]
print(players[0:3])
#Serão exibidos os itens na posição 0, 1 e 2.
```

```
['charles', 'martina', 'michael']
```

2. Delimitando ínicio e fim da fatia.

Podemos começar de qualquer índice.

Exemplo:

```
#Exibindo do segundo ao quarto item.
players = ["charles","martina","michael","florence","eli"]
print(players[1:4])
```

```
['martina', 'michael', 'florence']
```

3. Omitindo índices

Se omitirmos o primeiro índice, o Python começará do índice 0 (ínicio). De maneira analóga, se omitirmos o segundo índice (índice final), o Python terminará no último item.

Exemplo:

```
#Exibindo os 2 primeiros elementos de uma lista.
players = ["charles","martina","michael","florence","eli"]
print(players[:2])
```

```
['charles', 'martina']
```

4. índice negativo

O índice negativo devolve um elemento a determina distância do final da lista. Assim podemos exibir qualquer fatia a partir do final da lista.

Exemplo:

```
#Exibindo os 3 últimos elementos de uma lista.
players = ["charles","martina","michael","florence","eli"]
print(players[-3:])
```

```
['michael', 'florence', 'eli']
```

4.5.2 Percorrendo uma fatia com um laço - `for`

Podemos usar uma fatia em um laço `for` se quisermos percorrer um subconjunto de elementos de uma lista.

Exemplo:

```
players = ["charles","martina","michael","florence","eli"]
print("Here are the first three players on my team:")
for player in players[:3]:
  print(player.title())
```

```
Here are the first three players on my team:
Charles
Martina
Michael
```

4.5.3 Copiando uma lista

Vamos explorar o modo de copiar uma lista e analisar uma situação em que copiar uma lista é útil.

1. Copiando uma lista inteira, usando *fatia*.

Podemos criar uma fatia que inclua a lista inteira, omitindo o primeiro e segundo índices.

Exemplo:

```
#Usamos o metódo de fatia para copiar listas.
my_foods = ["pizza","falafel","carrot cake"]
friend_foods = my_foods[:]

print("My favorite food are:")
print(my_foods)

print("\nMy friend's favorite food are:")
print(friend_foods)
```

```
My favorite food are:
['pizza', 'falafel', 'carrot cake']

My friend's favorite food are:
['pizza', 'falafel', 'carrot cake']
```

Ambas as listas `my_foods` e `friend_fooods`, contém os mesmos elementos, porém são listas diferentes. Ao modificarmos uma delas a outra não é modificada automáticamente, por serem listas diferentes.

2. Variáveis que apontam para mesma lista.

Se ao invés de copiarmos uma fatia de uma lista para a outra, mesmo que seja a lista inteira, definirmos que uma variável é igual a outra, nesse caso criamos duas variáveis que apontam para a mesma lista. Ou seja, se modificarmos qualquer uma das listas, a outra é automaticamente modificada, pois ambas são a mesma lista.

Exemplo:

```
#Ambas variáveis apontam para a mesma lista.
my_foods = ["pizza","falafel","carrot cake"]
friend_foods = my_foods

friend_foods.append("ice cream")

print("My favorite food are:")
print(my_foods)

print("\nMy friend's favorite food are:")
print(friend_foods)
```

```
My favorite food are:
['pizza', 'falafel', 'carrot cake', 'ice cream']

My friend's favorite food are:
['pizza', 'falafel', 'carrot cake', 'ice cream']
```

4.6 Tuplas

Tuplas são listas em que os itens não são criadas para mudar (listas imutáveis).

4.6.1 Definindo uma tupla

Uma tupla se parece com uma lista, exceto por usar parênteses no lugar de colchetes.

Sintaxe:

```
tuplas = (valor_1,valor_2,valor_3,...)
```

Exibimos cada elemento de uma tupla com a mesma sintaxe que usamos para acessar elementos de uma lista.

Exemplo:

```
dimensions = (200,50)
print(dimensions[0])
```

```
200
```

Se tentarmos alterar algum elemento de uma tupla, será retornado um erro de tipo.

4.6.2 Percorrendo todos os valores de uma tupla com um laço

Podemos percorrer uma tupla usando um laço `for`, da mesma forma que uma lista.

Exemplo:

```
dimensions = (200,50)
for dimension in dimensions:
  print(dimension)
```

```
200
50
```

4.6.3 Sobrescrevendo uma tupla

Não é possível modificar os elementos de uma tupla. Retornaria um erro de tipo.

Esse tipo de operação não funcionaria:

`tupla[0] = valor_novo`

Porém é possível subrescrever a tupla, imputando novos valores a variável.

Exemplo:

```
#Sobrescrevendo uma tupla
dimensions = (200,50)
print("Original dimensions:")
for dimension in dimensions:
  print(dimension)

dimensions = (400,100)
print("\nModified dimensions:")
for dimension in dimensions:
  print(dimension)
```

```
Original dimensions:
200
50

Modified dimensions:
400
100
```

5 Estatística básica

5.1 Teoria

- Definição de Estatística:

  A Estatística de uma maneira geral compreende aos métodos científicos para COLETA, ORGANIZAÇÃO, RESUMO, APRESENTAÇÃO e ANÁLISE de Dados de Observação (Estudos ou Experimentos), obtidos em qualquer área de conhecimento. A finalidade é a de obter conclusões válidas para tomada de decisões.

  - Estatística Descritiva

    Parte responsável basicamente pela COLETA e SÍNTESE (Descrição) dos Dados em questão.

    Disponibiliza técnicas para o alcance desses objetivos. Tais Dados podem ser provenientes de uma AMOSTRA ou POPULAÇÃO.

  - Estatística Inferencial

    É utilizada para tomada de decisões a respeito de uma população, em geral fazendo uso de dados de amostrais.

    Essas decisões são tomadas sob condições de INCERTEZA, por isso faz-se necessário o uso da TEORIA DA PROBABILIDADE.

- O fluxograma da estatística descritiva pode ser espresso da seguinte forma:

Figure 1: Fluxograma da estatística descritiva.

- A representação tabular (**Tabelas de Distribuição de Frequências**) deve conter:
  - Cabeçalho

    Deve conter o suficiente para que as seguintes perguntas sejam respondidas "**o que?**" (Relativo ao fato), "**onde?**" (Relativo ao lugar) e "**quando?**" (Correspondente à época).
  - Corpo

    É o lugar da Tabela onde os dados serão registrados. Apresenta colunas e sub colunas.
  - Rodapé

    Local destinado à outras informações pertinentes, por exemplo a Fonte dos Dados.
- População e Amostras:
  - População

    É o conjunto de todos os itens, objetos ou pessoas sob consideração, os quais possuem pelo menos uma característica (variável) em comum. Os elementos pertencentes à uma População são denominados "Unidades Amostrais".
  - Amostras

    É qualquer subconjunto (não vazio) da População. É extraída conforme regras pré-estabelecidas, com a finalidade de obter "estimativa" de alguma característica da População.

- Tipos de variáveis

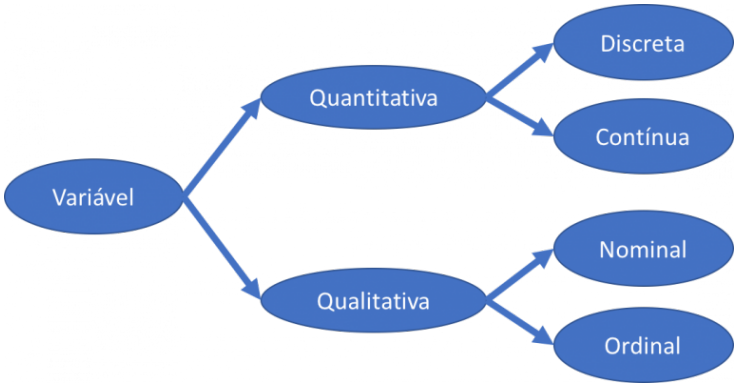


Figure 2: Tipos de variáveis.

  - *Qualitativo nominal*

    Não possuem uma ordem natural de ocorrência.

  - *Qualitativo ordinal*

    Possuem uma ordem natural de ocorrência.

  - *Quantitativo descreta*

    Só podem assumir valores inteiros, pertencentes a um conjunto finito ou enumerável.

  - *Quantitativo continua*

    Podem assumir qualquer valor em um determinado intervalo da reta dos números reais.

5.2 Preparação dos dados (sumariazar dados coletados)

- Frequência (conceito)

  É a quantidade de vezes que um valor é observado dentro de um conjunto de dado.

- Distribuição em frequências

  - A distribuição tabular é denominada: “Tabela de Distribuição de Frequências”.
  - Podemos separar em 3 modelos de distribuição tabular:
    * Variável Quantitativa Discreta.
    * Variável Quantitativa Contínua.
    * Variáveis Qualitativas.

5.2.1 Variável Quantitativa Discreta

- Passos da preparação dos dados:
  - 1º Passo - **DADOS BRUTOS**:
    Obter os dados da maneira que foram coletados.
  - 2º Passo - **ROL**:
    Organizar os DADOS BRUTOS em uma determinada ordem (crescente ou decrescente).
  - 3º Passo - **CONSTRUÇÃO TABELA**:
    Na primeira coluna são colocados os valores da variável, e nas demais as respectivas frequências.
  - Frequência absoluta simples.
    Nº de vezes que cada valor da variável se repete.
- Principais campos da **distribuição tabular de variaveis quantitativas discreta**:
  - n é o número total de elementos da amostra.
  - x_i é o número de valores distintos que a variavel assume.
  - F_i é a Frequência Absoluta Simples.
  - f_i é a Frequência Relativa Simples.
  - $f_i\%$ é a Frequência Relativa Simples Percentual. $f_i\% = f_i \cdot 100\%$.
  - F_a é a Frequência Absoluta Acumulada.

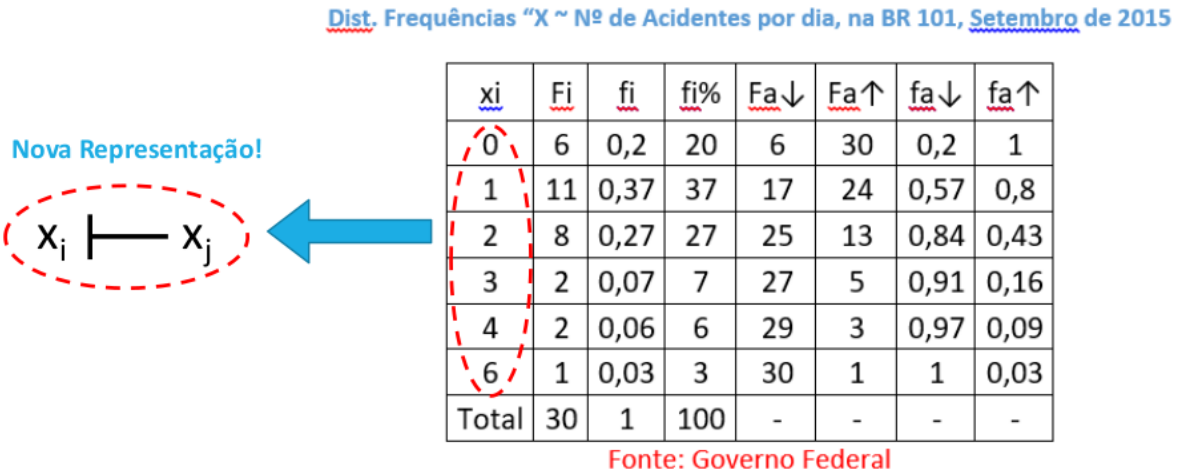
xi	Fi	fi	fi%	Fa↓	Fa↑	fa↓	fa↑
0	6	0,2	20	6	30	0,2	1
1	11	0,37	37	17	24	0,57	0,8
2	8	0,27	27	25	13	0,84	0,43
3	2	0,07	7	27	5	0,91	0,16
4	2	0,06	6	29	3	0,97	0,09
6	1	0,03	3	30	1	1	0,03
Total	30	1	100	-	-	-	-

Figure 3: Distribuição tabular quantitativa discreta.

Observação:
As setas simbolizam ordem crescente ou decrescente.

5.2.2 Variável Quantitativa Contínua

- Teoria:
  - A construção da representação tabular é realizada de maneira análoga ao caso das variáveis discretas.
  - As frequências são agrupadas em classes, denominadas de "Classes de Frequência".
  - Denominada "Distribuição de Frequências em Classes" ou "Distribuição em Frequências Agrupadas".



xi	Fi	fi	fi%	Fa↓	Fa↑	fa↓	fa↑
0	6	0,2	20	6	30	0,2	1
1	11	0,37	37	17	24	0,57	0,8
2	8	0,27	27	25	13	0,84	0,43
3	2	0,07	7	27	5	0,91	0,16
4	2	0,06	6	29	3	0,97	0,09
6	1	0,03	3	30	1	1	0,03
Total	30	1	100	-	-	-	-

Figure 4: Distribuição de frequências em classes.

- Convencionar o tipo de intervalo para as classes de frequência:

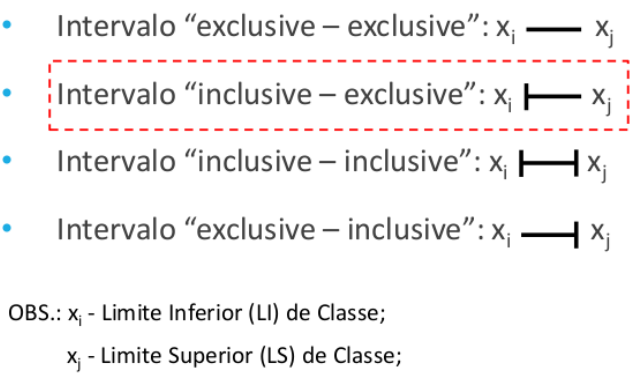


Figure 5: Intervalo de classes, distribuição de frequências quantitativa continua.

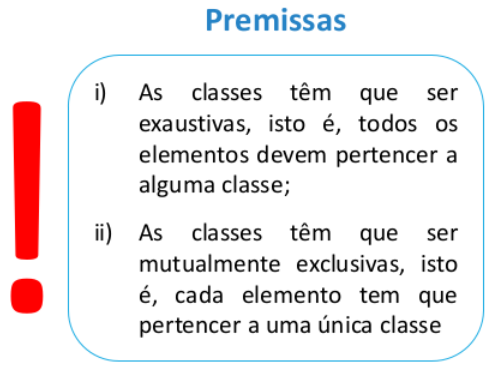


Figure 6: Distribuição frequências quantitativa continua, premissas.

Passos para contruir a **Tabela Distribuição de Frequências Contínua**:

1. Como estabelecer o **número de classes** (k):

- Normalmente varia de 5 a 20 classes.

- Critério fórmula de Sturges:

$$k \cong 1 + 3,3 \cdot \log(n)$$

  Onde n é o número de elementos amostrais.
  Arredondar k para número inteiro.

- Critério da Raiz quadrada:

$$k \cong \sqrt{n}$$

  Onde n é o número de elementos amostrais.
  Arredondar k para número inteiro.

2. Como calcular a **Amplitude Total** (AT_x):

- Diferença entre o maior e o menor valor observado.
- Intervalo de variação dos valores observados.
- Aproximar valor calculado para múltiplo do nº classes (k).
- Garantir inclusão dos valores mínimo e máximo.
- Cálculo:

$$AT_x = Mx(X_i) - Mn(X_i)$$

Onde,
AT_x é a Amplitude Total;
$Mx(X_i)$ é o *valor máximo das amostras*;
$Min(X_i)$ é o *valor mínimo das amostras*.

- Exemplo:
Se $k = 5$,
$AT_x = 28$
Logo, arredondando $AT_x = 30$, para aproximar o valor AT_x de um múltiplo de k.

3. Como cálcular a **Amplitude das classes da frequência** (h):

- As classes terão amplitudes iguais.
- Cálculo:

$$h = h_i = \frac{AT_x}{k}$$

Onde,
k é o **número de classes** e AT_x é a **Amplitude Total**.
h deve ser arredondado para cima, num número inteiro.

4. Como determinar o ponto médio das classes, representatividade da classe (p_i):

$$p_i = \frac{(LS_i - LI_i)}{2}$$

Onde,
LS_i é o limite superior da classe.
LI_i é o limite inferior da classe.

5. Passos da preparação dos dados:

- 1º Passo - **DADOS BRUTOS**:
  Obter os dados da maneira que foram coletados.
- 2º Passo - **ROL**:
  Organizar os DADOS BRUTOS em uma determinada ordem (crescente ou decrescente).
- 3º Passo - **CONSTRUÇÃO TABELA**:
  Na primeira coluna são colocados as classes, e nas demais as respectivas frequências.
- Exemplo:

Nº Classe	Classes (xi)	Fi	fi	fi%	Fa↓	Fa↑	fa↓	fa↑	fa↓%	pi
1	45 |--- 52	3	0,08	8	3	40	0,08	1	100	48,5
2	52 |--- 59	7	0,18	18	10	37	0,26	0,92	92	55,5
3	59 |--- 66	11	0,28	28	21	30	0,53	0,75	75	62,5
4	66 |--- 73	10	0,25	25	31	19	0,78	0,47	47	69,5
5	73 |--- 80	4	0,10	10	35	9	0,88	0,22	22	76,5
6	80 |--- 87	4	0,10	10	39	5	0,98	0,12	12	83,5
7	87 |--- 94	1	0,02	2	40	1	1,00	0,02	2	90,5
Total		40	1,00	100	-	-	-	-		-

Fonte: Dados Fictícios

Figure 7: Tabela de_distribuição de frequência quantitativa continua.

X_i são as classes.
F_i é a Frequência Absoluta Simples.
f_i é a Frequência Relativa Simples.
$f_i\%$ é a Fequência Relativa Simples Percentual.
F_a é a Frequência Absoluta Acumulada.
f_a é a Fequência Absoluta Acumulada Simples.
$f_a\%$ é a Fequência Absoluta Acumulada Simples Percentual.
p_i é a Representatividade da classe (ponto médio das classes).

5.2.3 Variáveis Qualitativas

- Passos da preparação dos dados:
  - Análogo ao procedimento para dados discretos.
  - 1º Passo - **DADOS BRUTOS**:
    Obter os dados da maneira que foram coletados.
  - 2º Passo - **ROL**:
    Nesse caso é feita organização dos DADOS BRUTOS em ordem (Crescente ou Decrescente) de importância.
  - 3º Passo - **CONSTRUÇÃO TABELA** (Com duas ou mais colunas).
- Distribuição de Frequencia:
  - x_i é o número de valores distintos que a variável assume.
  - F_i é a Frequência Absoluta Simples.
  - f_i é a Frequência Relativa Simples.
  - $f_i\%$ é a Fequência Relativa Simples Percentual.
  - Inserir comentário sobre os dados.

5.3 Medidas de posição

- Localizar a *maior concentração de valores* de uma distribuição.
- *Sintetizar o comportamento* do conjunto do qual ele é originário.
- Possibitar a *comparação* entre séries de dados.
- As principais **medidas de posição** são:
  - **Média Aritmética** (Simples e Ponderada)
  - **Mediana**
  - **Moda**
  - **Separatrizes**
- Medidas de posição comparação:

Medidas de Posição - Comparação

Medida	Definição	Vantagens	Desvantages
Média	Centro da Distribuição	Reflete todos os valores	É afetada por valores extremos
Mediana	Divide a distribuição ao meio	Menos sensível a valores extremos	Difícil determinar para grandes quantidades de dados
Moda	Valor mais frequente	Valor típico	Não é utilizado em análises matemáticas

5.3.1 Média Aritmética (Simples e Ponderada)

- **Média Aritmética Simples**, dados Não-Agrupados (não tabelados):
  - **Média Aritmética** ($\overline{x}$) é o valor médio dos dados da distribuição.
  - É a soma de todos os elementos, dividido pelo número total de elementos.
  - Cálculo:
$$\overline{x} = \frac{Soma}{n_{Total}}$$
- **Média Aritmética Ponderada**, dados Agrupados (tabelados):
  - Atribui-se um peso a cada valor da série.
  - É o *Ponto Médio das Classes* (p_i), multiplicado por suas respectivas *Frequência Absoluta Simples* (F_i), somadas. Dividido pelo *Número Total de Elementos da Amostra* (n).
  - Cálculo:
$$\overline{x} = \frac{\sum_{i=1}^{n} p_i \cdot F_i}{n_{Total}}$$
    ou,
$$\overline{x} = \frac{(p_1 \cdot F_1) + (p_2 \cdot F_2) + (p_3 \cdot F_3) + ...}{n_{Total}}$$

5.3.2 Mediana ($md(x)$)

5.3.2.1 Mediana Discreta

- Com dados em ROL, é o valor que divide o conjunto de dados em duas partes iguais.
- No caso de número de elementos impar, a mediana ($md(x)$) é o elemento central.
- No caso de número de elementos par, a mediana ($md(x)$) é a média aritmética simples dos valores centrais:

$$md(x) = \frac{x_{\frac{n}{2}} + x_{\frac{n+1}{2}}}{2}$$

Onde,
x é a posição do elemento;
n é o número total de elementos.

5.3.2.2 Mediana Contínua

- Mediana (md) em distribuição de frenquência em variável contínua (dados agrupados em classes):

1. Fazer a coluna da **Frequência Absoluta Acumulada**, que é o somatório das frequências ao logo das classes.
2. Definindo o **Intervalo da Mediana**.

- Obter o número total de elementos n (somatório das frenquências de classes),

$$n = \sum f_i$$

- Determinar a posição do elemento do meio do somatório das frequencias:

$$x = \frac{\sum f_i}{2}$$

- A classe que contém essa posição x na **Frequência Absoluta Acumulada** é a classe do *intervalo da mediana*.

3. Cálculo da Mediana:

$$md = Li + (\frac{\frac{\sum fi}{2} - Fa_{anterior}}{f_{intervalo}} \cdot h)$$

Onde,
Li é o limite inferior do *intervalo da mediana*;
$\sum fi$ é o somatório das frequências (**frequência total** (n));
$Fa_{anterior}$ é a **Frequência Absoluta Acumulada** da classe anterior (linha anterior ao *intervalo da mediana*);
$f_{intervalo}$ é a **Frequência Absoluta Simples** do *intervalo da mediana*;
h é a Amplitudade da classe do *intervalo da mediana*.

$$h = Ls - Li$$

5.3.3 Moda

- Moda ou $Mo(x)$: Valor com maior frequência de ocorrência em uma distribuição.
- Podem haver mais de um valor distinto com maior frequência, podendo assim ter mais de um valor na moda.
- Moda com frequência Continua:

1. **Moda Bruta** (M_{Bruta}):

– Achar a classe com maior frequência, esse será o *Intervalo Modal*.
– Cálcular o *Ponto Médio* (Representatividade da classe) do *Intervalo Modal*:

$$PM = \frac{LS + LI}{2}$$

Onde,

LS = Limite superior da classe;

LI = Limite inferior da classe.

– O *Ponto Médio* do *Intervalo Modal* será a **Moda Bruta**(M_{Bruta}).

2. **Moda King** ou **Moda do Rei** (M_{King}):

– Determinar o intervalo (classe) com maior frequência, esse será o *Intervalo Modal*.
– Cálculo da Moda de King (M_{King}):

$$M_{King} = LI + (\frac{F_{post}}{F_{post} + F_{ant}} \cdot h)$$

Onde,

LI é o limite inferior da classe do *Intervalo Modal*;

F_{post} é a frequência da classe posterior ao *Intervalo Modal*;

F_{ant} é a frequência da classe anterior ao *Intervalo Modal*;

h é a amplitude do intervalo da classe

$$h = LS - LI$$

3. **Moda de Czuber** (M_{Czuber}):

– Determinar o intervalo (classe) com maior frequência, esse será o *Intervalo Modal*.

– Cálculo da **Moda de Czuber** (M_{Czuber}):

$$M_{Czuber} = LI + (\frac{\Delta_{ant}}{\Delta_{ant} + \Delta_{post}} \cdot h)$$

Onde,

LI é o limite inferior da classe do *Intervalo Modal*;

Δ_{ant} é a variação (diferença) da frequência da classe anterior (ao *Intervalo Modal*) com o *Intervalo Modal* (classe com maior frequência)

$$\Delta_{ant} = |F_i - F_{i-1}|$$

Δ_{post} é a variação (diferença) da frequência da classe posterior (ao *Intervalo Modal*) com o *Intervalo Modal* (classe com maior frequência)

$$\Delta_{ant} = |F_i - F_{i+1}|$$

h é a amplitude do intervalo da classe

$$h = LS - LI$$

5.3.4 Separatrizes

- **Separatrizes** são valores da distribuição que a dividem em partes quaisquer.
- A **mediana**, apesar de ser uma medida de tendência central, é também uma **separatriz** de ordem 1/2, ou seja, divide a distribuição em duas partes iguais.
- As **separatrizes** mais comumente usadas são:
  - **Quartis**
    Dividem a distribuição em quatro partes iguais, de ordem 1/4.
  - **Decis**
    Dividem a distribuição em 10 partes iguais, de ordem 1/10.
  - **Centis**
    Dividem a distribuição em 100 partes iguais, de ordem 1/100.
- Fórmula das Separatrizes:

1. Achar o **Intervalo da separatriz**

- É a classe em que se encontra a separatriz procurada.
- Fazer a coluna de **Frequencia Absoluta Acumulada** (F_a).
- É o somatório das frequencias (total das frequencias), multiplicado pela fração da separatriz procurada (k). O resultado é a posição da frequencia na coluna **Frequencia Absoluta Acumulada** (F_a).

$$P_k = k \cdot \sum f_i$$

A classe na qual a posição pertence é o **Intervalo da separatriz**.

2. Cálculo da separatriz:

$$Sp = L_i + (\frac{k \cdot \sum f_i - Fa_{anterior}}{f_{Intervalo}} * h)$$

Onde,
L_i é o limite inferior do **Intervalo da separatriz**;
k é a fração (porcentagem) da separatriz procurada;
$\sum f_i$ é o somatório das frequências;
$Fa_{anterior}$ é a **Frequência Absoluta Acumulada** da classe anterior ao **intervalo da separatriz**;
$f_{Intervalo}$ é a **Frequência Absoluta Simples** do **intervalo da separatriz**;
h é a **Amplitude** da classe (limite superior - limite inferior da classe).

$$h = Ls - Li$$

3. Cálculo de **Amplitude Interquartil** (AI):

- É a diferença entre 3º quartil e o 1º quartil.

$$AI = Q_3 - Q_1$$

- Para descobrir os valores dos Quartis (Q_1 e Q_3) basta usar o *cálculo das separatrizes*.

5.4 Medidas de dispersão

- Medem o grau de **variabilidade** (dispersão) dos valores observados em torno da **Média Aritmética**.
- Caracterizam a **representatividade da média** e o nivel de **homogeneidade** ou **heterogeneidade** dentro de cada grupo analizado.

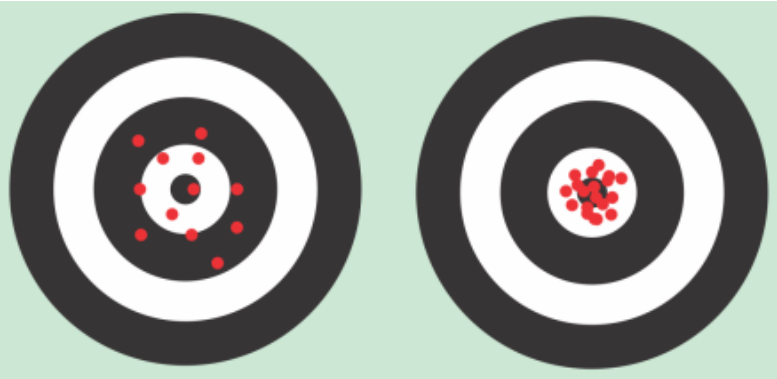


Figure 8: Exemplo de dispersão com heterogeneidade e homogeneidade.

5.4.1 Amplitude Total (A_T)

- Diferença entre o maior e o menor dos valores da série.
- Não considera a dispersão dos valores internos, apenas os extremos.
- Utilização limitada enquanto medida de dispersão, oferece pouca informação.
- Cálculo:

$$A_T = X_{Mx} - X_{Mn}$$

Onde,
X_{Mx} é o valor máximo da série;
X_{Mn} é o valor mínimo da série.

5.4.2 Desvio

5.4.2.1 Desvio Absoluto (D)

- Para dados não agrupados:
  - Os **Desvios Absolutos** (D) são a diferença absoluta entre um valor observado e a média aritmética:

    $$D = |x_i - \bar{X}|$$

    Onde,
    x_i é o **valor de cada elemento**;
    $\bar{x}$ é a **Média Aritmética**.
    Os **Desvios Absolutos** (D) são um conjunto de elementos como resposta final.

- Para dados agrupados, sem intervalo de classe:
  - Cálculo:

    $$d_i = |x_i - \bar{X}|$$

    Onde,
    x_i é o valor da variável discreta;
    $\bar{X}$ é a **Média Aritmética**.

- Para dados agrupados, com intervalo de classe:
  - Cálculo:

    $$d_i = |p_i - \bar{x}|$$

    Onde,
    p_i é a **Representatividade da classe** (ponto médio da classe);
    $\bar{x}$ é a **Média Aritmética Ponderada** cálculada para *dados agrupados continuos*:

    $$\bar{x} = \frac{\sum_{i=1}^{N} p_i \cdot f_i}{\sum f_i}$$

    É o *Ponto Médio das Classes* (p_i), multiplicado por suas respectivas *Frequência Absoluta Simples* (F_i), somadas. Dividido pelo *Número Total de Elementos da Amostra* (n).

5.4.2.2 Desvio Absoluto Médio (dm)

- É a **Média** dos **Desvios**.
- Para dados não agrupados:
  - Cálculo:

$$dm(x) = \frac{\sum_{i=1}^{n} |x_i - \bar{x}|}{n}$$

    Onde,
    x_i é o **valor de cada elemento**;
    $\bar{x}$ é a **Média Aritmética**;
    n é o **número total de elementos** (frequencia total).

- Para dados agrupados, sem intervalo de classe:
  - Cálculo:

$$D_M = \frac{\sum |d_i| \cdot f_i}{n}$$

    Onde,
    d_i é o **Desvio Absoluto** para dados agrupados, sem intervalo de classe;
    f_i é a **Frequência** de cada variável discreta;
    n é o número total de elementos (ou somatório das frequências).

- Para dados agrupados, com intervalo de classe:
  - Cálculo:

$$D_M = \frac{\sum |d_i| \cdot f_i}{\sum f_i}$$

    Onde,
    d_i é o **Desvio Absoluto** para dados agrupados, com intervalo de classe;
    f_i é a **frequência** de cada intervalo de classe.

5.4.3 Variância (σ^2 ou S^2)

- Leva em consideração os valores extremos e também os valores intermediários.
- Relaciona os desvios em torno da média (distancias dos valores ate a média).
- Média Aritmética dos quadrados dos desvios.
- O símbolo para **Variância Populacional** é o sigma ao quadrado (σ^2), já o símbolo para **Variância Amostral** é o "S" maiusculo ao quadrado (S^2).
- Cálculo para dados não agrupados:
  - População

    $$\sigma^2 = \sum_{i=1}^{N} \frac{(x_i - \bar{x})^2}{N}$$

    Onde,
    x_i é o valor de **cada elemento da série**;
    $\bar{x}$ é o valor da **Média Aritmética Simples**;
    N é o **número total da população**.
  - Amostra

    $$S^2 = \sum_{i=1}^{n} \frac{(x_i - \bar{x})^2}{n-1}$$

    Onde,
    x_i é o valor de **cada elemento da série**;
    $\bar{x}$ é o valor da **Média Aritmética Simples**;
    n é o **número de elementos da Amostra**;
    $(n-1)$ é por ser uma estimativa no caso da Amostra, trabalhando assim com um grau a menos de liberdade.
- Cálculo dados agrupados:
  - Para dados agrupados, sem intervalo de classe (**Variáveis Discretas**):
    - População

      $$\sigma^2 = \frac{\sum (x_i - \bar{X})^2 \cdot f_i}{\sum f_i}$$

      Onde,
      x_i é o valor de **cada elemento da série**;
      $\bar{X}$ é o valor da **Média Aritmética Ponderada**;
      f_i é a **Frequência** da variável;
      $\sum f_i$ é o somatório das **Frequências**.

* Amostra

$$S^2 = \frac{\sum (x_i - \bar{X})^2 \cdot f_i}{n-1}$$

Onde,
x_i é o valor de **cada elemento da série**;
$\bar{X}$ é o valor da **Média Aritmética Ponderada**;
f_i é a **Frequência** da variável;
$n-1$ ou $\sum f_i - 1$ é o somatório das **Frequências** da Amostra menos 1.

– Para dados agrupados, com intervalo de classe (**Variáveis Contínuas**):

* População

$$\sigma^2 = \frac{\sum (p_i - \bar{X})^2 \cdot f_i}{\sum f_i}$$

Onde,
p_i é a **Representatividade das Classe** (**Ponto Médio das Classes**);
$\bar{X}$ é o valor da **Média Aritmética Ponderada**;
f_i é a **Frequência** da variável;
$\sum f_i$ é o somatório das **Frequências**.

* Amostra

$$S^2 = \frac{\sum (p_i - \bar{X})^2 \cdot f_i}{n-1}$$

Onde,
p_i é a **Representatividade das Classe** (**Ponto Médio das Classes**);
$\bar{X}$ é o valor da **Média Aritmética Ponderada**;
f_i é a **Frequência** da variável;
$n-1$ ou $\sum f_i - 1$ é o somatório das **Frequências** da Amostra menos 1.

5.4.4 Desvio-padrão (σ ou S)

5.4.4.1 Variância x Desvio-padrão

- **Variância**:
  - Número em unidade "quadrada".
  - Maior dificuldade de compreensão e menor utilidade na estatística descritiva.
  - Extremamente relevante na inferência estatística e em combinações de amostras.
- **Desvio-padrão**:
  - Mais usado na comparação de diferenças entre conjuntos de dados.
  - Determina a dispersão dos valores em relação a **Média**.
  - Volta-se com os dados para a unidade original.

5.4.4.2 Desvio-padrão (Populacional e Amostral)

- Determina a dispersão dos valores em relação a **Média**.
- População

$$\sigma = \sqrt{\sigma^2}$$

  Onde,
  σ^2 é a **Variância Populacional**;
  σ é o **Desvio-padrão Populacional**.
- Amostra

$$S = \sqrt{S^2}$$

  Onde,
  S^2 é a **Variância Amostral**;
  S é o **Desvio-padrão Amostral**.

5.4.5 Coeficiente de Variação (CV)

5.4.5.1 Teoria

- Medida relativa de dispersão.
- Útil para comparação em termos relativos do grau de concentração.
- O **Coeficiente de Variação** (CV) é expresso em porcentagens.
- Diz-se que uma distribuição:
  - $CV \leq 15\%$ tem **Baixa Dispersão**.
  - $15\% < CV < 30\%$ tem **Média Dispersão**.
  - $CV \geq 30\%$ tem **Alta Dispersão**.

5.4.5.2 Cálculo do Coeficiente de Variação

- População:

$$CV = \frac{\sigma}{\bar{X}} \times 100$$

  Onde,
  σ é o **Desvio-padrão Populacional**;
  $\bar{X}$ é a **Média Populacional**.

- Amostra:

$$CV = \frac{S}{\bar{x}} \times 100$$

  Onde,
  S é o **Desvio-padrão Amostral**;
  $\bar{x}$ é a **Média Amostral**.

6 Análise Estatística

- Para fazer uma Análise Estatística eficiente de dados, necessitamos:
  - Limpar os dados
    Remover os *OUTLIER* (valores atipicos, inconsistentes).
  - Aplicar Estatística Descritiva aos dados
    As medidas de posição (**Média**, **Mediana** e **moda**) e dispersão (**Amplitude Total**, **Desvio**, **Desvio Médio**, **Variância**, **Desvio-padrão** e **Coeficiente de Variação**) são maneiras de descrever os dados.
  - Comparar as medidas dos dados
    Principalmente medidas de dispersão, me especial **Coeficiente de Variação**, são ótimas para comparar dados.
  - Previsão de dados
    A principal técnica é de **Regressão**, porém para aplicar, necessita que os dados estejam limpos e com pouca dispersão (quanto menor, melhor).

7 Instruções `IF`

Instruções `IF` são testes condicionais.

7.1 Testes condicionais

O serne da instrução `IF` esta uma expressão que deve ser avaliada como `True` ou `False`, chamado teste condicional. Esse teste decide se a instrução deve ser executada.

Teste condicional com resultado `True`, o código dentro do `IF` será executado.

Teste condicional com resultado `False`, o código dentro do `IF` não será executado.

7.2 Operações lógicas

A Table 13 apresenta as principais operações lógica do python. As operações lógicas retornam `True` ou `False`. A Table 14 mostra exemplos das operações lógicas.

Table 13: Operações Lógicas

Operação	Nome	Função
==	Igual a	Varifica se um valor é igual ao outro.
!=	Diferente de	Varifica se um valor é diferente ao outro.
>	Maior que	Varifica se um valor é maior que outro.
>=	Maior ou igual	Varifica se um valor é maior ou igual a outro.
<	Menor que	Varifica se um valor é menor que outro.
<=	Menor ou igual	Varifica se um valor é menor ou igual a outro.
and	E	Retorna True se ambas as afirmações forem verdadeiras.
or	Ou	Retorna True se uma das afirmações for verdadeiras.
not	Negação	Retorna Falso se o resultado for verdadeiro, ou o contrario.

Table 14: Operações Lógicas Exemplos

Operação	Exemplo
==	1==1 = True
!=	1!=2 = True
>	5>1 = True
>=	5>=5 = True
<	1<5 = True
<=	1<=4 = True
and	(1==1) and (4<5) = True
or	(1==1) or (2<1) = True
not	not (1==1) = False

Observações:
- Não confundir `=` com `==`. O sinal de `=` simples é uma atribuição de valor, enquanto que o sinal `==` duplo representa "igual a", sendo um operador lógico.
- Os operadores lógicos de igualdade (`==` e `!=`) fazem distinção entre letras maiúsculas e minúsculas.

7.3 Testando várias condições

Podemos testar duas (ou mais) condições ao mesmo tempo. Para isso as palavras reservadas `and` e `or` ajudam nesse tipo de situação.

7.3.1 Testando várias condições lógicas - `AND`

O operador lógico `and` nada mais é do que o `E` da lógica, então podemos comparar duas operações lógicas e compara-las seguindo a ideia da tabela verdade do operador `E`.

TABELA VERDADE - AND		
A	**B**	**A.B**
0	0	0
0	1	0
1	0	0
1	1	1

Figure 9: Tabela verdade do operador `AND`.

Exemplo:

```
age_0 = 22
age_1 = 18
print(age_0 >= 21 and age_1 >= 21)
age_1 = 22
print(age_0 >= 21 and age_1 >= 21)
```

```
False
True
```

7.3.2 Testando várias condições lógicas - `OR`

O operador lógico `or` nada mais é do que o `OU` da lógica, então podemos comparar duas operações lógicas e compara-las seguindo a ideia da tabela verdade do operador `OU`.

TABELA VERDADE - OR		
A	**B**	**A+B**
0	0	0
0	1	1
1	0	1
1	1	1

Figure 10: Tabela verdade do operador `OR`.

Exemplo:

```
age_0 = 22
age_1 = 18
print(age_0 >= 21 or age_1 >= 21)
age_0 = 18
print(age_0 >= 21 or age_1 >= 21)
```

```
True
False
```

7.4 Verificando se um valor está em uma lista - `IN`

Para descobrir se um valor em particular já esta em uma lista, utilizamos a palavra reservada `in`.

Exemplo:

```
requested_toppings = ['mushrooms','onions','pineapple']
print('mushrooms' in requested_toppings)
print('pepperoni' in requested_toppings)
```

```
True
False
```

7.5 Verificando se um valor não está em uma lista - `NOT IN`

Para descobrir se um valor em particular não esta em uma lista, utilizamos a palavra reservada `not in`.

Exemplo:

```
banned_users = ['andrew','carolina','david']
user = 'marie'

if user not in banned_users:
  print(user.title() + ", you can post a response if you wish.")
```

```
Marie, you can post a response if you wish.
```

7.6 Expressões booleanas

Um valor booleano é `True` ou `False`, exatamente como o valor de uma expressão condicional após ter sido avaliada.
Valores booleanos muitas vezes são usados para manter o controle de terminada condição.

```
game_active = True
can_edit = True
```

7.7 Instruções `IF`

Testes condicionais fazem parte das instruções `if`. Há vários tipos de instruções `if`, a escolha depende de quantas condições precisam ser testadas.

Os próximos subtópicos são as possibilidades de instruções `if`.

7.7.1 Instruções `if` simples

A instrução `if` mais simples contém um teste e uma ação.
Sintaxe:

```
if teste_condicional:
  ação
```

Ao avaliar o teste condicional e o resultado for `True`, as ações contidas dentro do `if` são executadas, caso contrario a ações contidas dentro da instrução `if` não são executadas.

Exemplo:

```
age = 19
if age >= 18:
  print("You are old enough to vote!")
```

```
You are old enough to vote!
```

7.7.2 Instruções `if-else`

Um bloco `if-else` é semelhante a uma instrução `if` simples, porém a instrução `else` permite definir ação ou um conjunto de ações executado quando o teste condicional falhar.
Sintaxe:

```
if teste_condicional:
  Ação_True
else:
  Ação_False
```

Exemplo:

```
age = 17
if age >= 18:
  print("You are old enough to vote!")
else:
  print("Sorry, you are too young to vote.")
```

```
Sorry, you are too young to vote.
```

7.7.3 Sintaxe `if-elif-else`

Muitas vezes se precisará testar mais de duas situações possíveis, para isso é usado a sintaxe `if-elif-else`. O Python executará apenas um bloco em uma cadeia `if-elif-else`. Cada bloco é executado em sequência, ate que algum deles passe. Quando um teste passar, o código após esse teste será executado e o Python ignorará o restante dos testes.
Sintaxe:

```
if teste_condicional_1:
  Ação_teste_1
elif teste_condicional_2:
  Ação_teste_2
else:
  Ação_3
```

Exemplo:

```
age = 12
if age < 4:
  print("You admission cost is $0.")
elif age < 18:
  print("You admission cost is $5.")
else:
  print("You admission cost is $10.")
```

```
You admission cost is $5.
```

7.7.4 Usando vários blocos `elif`

Podemos usar quantos blocos `elif` quisermos em nosso código.
Sintaxe:

```
if teste_condicional_1:
  Ação_teste_1
elif teste_condicional_2:
  Ação_teste_2
elif teste_condicional_3:
  Ação_teste_3
elif teste_condicional_4:
  Ação_teste_4
else:
  Ação_5
```

Exemplo:

```
age = 12
if age < 4:
  price = 0
elif age < 18:
  price = 5
elif age >= 65:
  price = 5
else:
  price = 10
print("Your admission cost is $" + str(price) + ".")
```

```
Your admission cost is $5.
```

7.7.5 Omitindo o bloco `else`

Python não exige um bloco `else` no final de uma cadeia `if-elif`. As vezes um bloco `else` é útil, outras vezes, é mais claro usar uma instrução `elif` adicional que capture a condição específica de interesse. `else` é uma função que captura tudo. Ela corresponde a qualquer condição não atendida por teste `if` ou `elif` específicos e isso, ás vezes, pode incluir dados inválidos ou maliciosos. É uma boa prática considerar usar um último bloco `elif` e omitir o bloco `else`.
Sintaxe:

```
if teste_condicional_1:
  Ação_teste_1
elif teste_condicional_2:
  Ação_teste_2
elif teste_condicional_3:
  Ação_teste_3
elif teste_condicional_4:
  Ação_teste_4
```

Exemplo:

```
age = 12
if age < 4:
  price = 0
elif age < 18:
  price = 5
elif age >= 65:
  price = 5
elif age < 65:
  price = 10

print("Your admission cost is $" + str(price) + ".")
```

```
Your admission cost is $5.
```

7.7.6 Testando várias condições

A cadeia `if-elif-else` é eficaz, mas é apropriada somente quando você quiser que apenas um teste passe. assim que encontrar um teste que passe, o interpretador Python ignorará o restante dos testes.
As vezes, porém, é importante verificar todas as condições de interesse. Nesse caso, podemos usar um série de instruções `if` simples, sem blocos `elif` ou `else`.
Em suma, se quiser que apenas um bloco de código seja executado, utilize uma cadeia `if-elif-else`. Se mais de um bloco de código deve ser executado, utilize uma série de instruções `if` independentes.
Sintaxe:

```
if teste_condicional_1:
  Ação_teste_1
if teste_condicional_2:
  Ação_teste_2
if teste_condicional_3:
  Ação_teste_3
if teste_condicional_4:
  Ação_teste_4
```

Exemplo:

```
#Pizzaria
requested_toppings = ['mushrooms','extra cheese']

if 'mushrooms' in requested_toppings:
  print("Adding mushrooms.")
if 'pepperoni' in requested_toppings:
  print("Adding pepperoni.")
if 'extra cheese' in requested_toppings:
  print("Adding extra cheese.")

print("\nFinished making your pizza!")
```

```
Adding mushrooms.
Adding extra cheese.

Finished making your pizza!
```

7.8 Usando instruções `if` com listas

Algumas tarefas interessantes podem ser feitas se combinarmos listas com instruções `if`. Podemos prestar atenção em valores especiais, que devem ser tratados de modo diferente de outros valores da lista.

7.8.1 Verificando itens especiais

Podemos dar tratamento especial à determinado item de uma lista, criando um bloco especial de ação para ele.
Exemplo de pizzaria de como tratar itens especiais:

```
request_toppings = ['mushrooms','green peppers','extra cheese']
for request_topping in request_toppings:
  if request_topping == 'green peppers':
    print("Sorry, we are out of green peppers right now.")
  else:
    print("Adding " + request_topping + ".")
print("\nFinished making your pizza!")
```

```
Adding mushrooms.
Sorry, we are out of green peppers right now.
Adding extra cheese.

Finished making your pizza!
```

7.8.2 Varificando se uma lista não esta vazia

Os usuários podem fornecer informações a serem armazenadas em uma lista, por isso não podemos supor que a lista não seja vazia. Nessa situação é conveniente testar se uma lista não esta vazia antes de executar um laço.
Quando o nome de uma lista é usado em uma instrução `if`, o Python devolve `True` se a lista contiver pelo menos um item; Uma lista vazia é avaliada como `False`.

Exemplo:

```
requested_toppings = []
if requested_toppings:
  for requested_topping in requested_toppings:
    print("Adding "+resquested_topping+".")
    print("\nFinished making your pizza!")
else:
  print("Are you sure you want a plain pizza?")
```

```
Are you sure you want a plain pizza?
```

7.8.3 Usando várias listas

Ao útilizar mais de uma lista, podemos usar listas e instruções `if` para garantir que o dado de entrada faça sentido antes de atuar sobre ele.
Um lista pode ser fechada (tupla) e representar o estoque da loja e outra lista o pedido do cliente. Assim teriamos que verificar o que bate e o que não bate entre as duas listas.

Exemplo:

```
available_toppings = ('mushrooms','olives','green peppers','pepperoni','pineapple','extra ch

requested_toppings = ['mushrooms','french fries','extra cheese']

for requested_topping in requested_toppings:
  if requested_topping in available_toppings:
    print("Adding " + requested_topping + ".")
  else:
    print("Sorry, we don't have " + requested_topping + ".")
print("\nFinished making your pizza!")
```

```
Adding mushrooms.
Sorry, we don't have french fries.
Adding extra cheese.

Finished making your pizza!
```

8 Dicionários

8.1 Dicionário simples

Os Dicionários permitem conectar informações relacionadas.
Sintaxe:

```
nome_dicionario = {'chave_1': 'valor_1', 'chave_2':'valor_2', ...}
```

Exemplo:

```
alien_0 = {'color':'green','points': 5}

print(alien_0['color'])
print(alien_0['points'])
```

```
green
5
```

8.2 Trabalhando com dicionários

Um dicionário em Python é uma coleção de `chave-valor`. Cada `chave` é conectada a um `valor`, e podemos usar a `chave` para acessar o `valor` associado a ela.
O `valor` pode ser um número, uma string, uma lista, ou até outro dicionário.
Em Python o dicionário é apresentado entre chaves {}, com uma série de pares `chave-valor` entre elas.
Exemplo:

```
alien_0 = {'color':'green','points': 5}
```

Um par `chave-valor` é um conjunto de valores associados um ao outro. Quando fornecemos uma `chave`, Python devolve o `valor` associado a essa `chave`. Toda `chave` é associada a seu `valor` por meio de dois-pontos, e pares `chave-valor` individuais são separados por vírgula.
Podemos armazenar quantos pares `chave-valor` quisermos em um dicionário.

8.3 Acessando valores em um dicionário

Para obter o `valor` associado a uma `chave`, especifique o nome do dicionário e coloque a chave entre colchetes, como a seguir:

```
alien_0 = {'color':'green'}
print(alien_0['color'])
```

```
green
```

Essa instrução devolve o `valor` associado a `chave` 'color' do dicionário `alien_0`.

Como podemos ter um número ilimitado de de pares de `chave-valor` em um dicionário, para acessar o valor de interesse basta colocar o nome da `chave` cujo o `valor` queremos acessar.

Exemplo:

```
alien_0 = {'color':'green','points': 5}
new_points = alien_0['points']

print("You just earned " + str(new_points) + " points!")
```

```
You just earned 5 points!
```

Lembrando que números para serem plotados em tela precisam ser tranformados em strings, através da função `str()`.

8.4 Adicionando novos pares `chave-valor`

Dicionários são estruturas dinâmicas, e você pode adicionar novos pares `chave-valor` em um dicionário a qualquer momento. Por exemplo, para acrescentar um novo par `chave-valor`, especifique o nome do dicionário, seguido da nova chave entre colchetes, justamente com o novo valor.
Exemplo:

```
alien_0 = {'color':'green','points': 5}
print(alien_0)

alien_0['x_position'] = 0
alien_0['y_position'] = 25
print(alien_0)
```

```
{'color': 'green', 'points': 5}
{'color': 'green', 'points': 5, 'x_position': 0, 'y_position': 25}
```

A versão final do dicionário contém quatro pares `chave-valor`. Dois pares originais especificam a cor e o valor da pontuação, enquanto os dois pares adicionais especificam a posição do alienígena.
Oberserve que a ordem dos pares `chave-valor` não coincidem com a ordem em que foram adicionados. O Python não se importa com a ordem em que armazenamos cada par `chave-valor`, ele só se importa com a conexão entre cada `chave` e seu `valor`.

8.5 Dicionário vazio

As vezes, é conveniente ou até mesmo necessário começar com um dicionário vazio e então acrescentar novos itens a ele. Para começar a preencher um dicionário vazio, defina-o com um conjunto de `chaves` vazio e depois acrescentar cada para par `chave-valor` em sua própria linha.
Exemplo:

```
alien_0 = {}

alien_0['color'] = 'green'
alien_0['points'] = 5

print(alien_0)
```

```
{'color': 'green', 'points': 5}
```

Nesse caso, definimos um dicionário `alien_0` vazio e, em seguida, adicionamos valores para cor e pontuação.

Geralmente usamos dicionários vazios quando armazenamos dados fornecidos pelo usuário em um dicionário, ou quando escrevemos um código que gere um grande número de pares `chave-valor` automaticamente.

8.6 Modificando valores em um dicionário

Para modificar um `valor` em um dicionário, especifique o nome do dicionário com a `chave` entre conchetes e o novo `valor` que você quer associar a essa `chave`.
Exemplo:

```
alien_0 = {'color':'green'}
print("The alien is " + alien_0['color'] + ".")

alien_0['color'] = 'yellow'
print("The alien is now " + alien_0['color'] + ".")
```

```
The alien is green.
The alien is now yellow.
```

Inicialmente, definimos um dicionário para `alien_0` que contém apenas a cor do alienigena. Em seguida, modificamos o `valor` associado a `chave` 'color' para 'yellow'.

8.7 Removendo pares `chave-valor`

Quando não houver mais necessidade de uma informação armazenada em um dicionário, podemos usar a instrução `del` para remover totalmente um par `chave-valor`.
Tudo que `del` precisa é do nome do dicionário e da `chave` que você deseja remover.
Exemplo:

```
alien_0 = {'color':'green','points':5}
print(alien_0)

del alien_0['points']
print(alien_0)
```

```
{'color': 'green', 'points': 5}
{'color': 'green'}
```

A instrução `del` diz ao Python para apagar a `chave` 'points' do dicionário `alien_0` e remover o `valor` associado a essa `chave` também. A saída mostra que a `chave` 'points' e seu `valor` igual a 5 foram apagados, porém o restante do dicionário não foi afetado.

8.8 Dicionário de objetos semelhantes

O exemplo anterior envolveu a armazenagem de diferentes tipos de informação sobre o mesmo objeto: um alienigena em um jogo. Também podemos usar um dicionário para armazenar um tipo de informação sobre vários objetos.
Por exemplo, suponha que você queira fazer uma enquete com várias pessoas e perguntar-lhes qual é a sua linguagem de programação favorita.
Exemplo:

```
favorite_languages = {
  'jen':'python',
  'sarah':'c',
  'edward':'ruby',
  'phil':'python'
  }

print("Sarah's favorite language is "+
    favorite_languages['sarah'].title()+
    ".")
```

```
Sarah's favorite language is C.
```

Esse exemplo também mostra como podemos dividir uma instrução longa em várias linhas, indentando as linhas e usando algum operador como parâmetro para finalizar uma linha. No caso do `print` o operador de concatenação (+), no caso do dicionário a vírgula.

8.9 Percorrendo um dicionário com um laço

Como um dicionário pode conter uma grande quantidade de dados, Python permite percorrer um dicionário com um laço. Dicionários podem ser usados para armazenar informações de várias maneiras. Assim, há diversos modos diferentes de percorrê-los com um laço. Podemos percorrer todos os pares `chave-valor` de um dicionário usando suas `chaves` ou seus `valores`.

8.9.1 Percorrendo todos os pares `chave-valor` com um laço

Se quisermos ver tudo que está armazenado no dicionário, podemos percorrer o dicionário com um laço `for`.
Exemplo:

```
user_0 = {
  'username':'efermi',
  'first':'erico',
  'last':'fermi'
}
for key, value in user_0.items():
  print("\nKey: " + key)
  print("Value: " + value)
```

```
Key: username
Value: efermi

Key: first
Value: erico

Key: last
Value: fermi
```

Para escrever um laço `for` paraum dicionário, devemos criar nomes para as duas variáveis que armazenarão a `chave` e o `valor` de cada par `chave-valor`. Podemos escolher qualquer nome que quisermos para essas duas variáveis.
A instrução `for` inclui o nome do dicionário, seguido do método `items()`, que devolve uma lista de pares `chave-valor`. O laço `for` então armazena cada um desses pares nas duas variáveis especificadas.
Observe que os pares `chave-valor` não são devolvidos na ordem em que foram armazenados, mesmo quando percorremos o dicionário com um laço. O Python não se importa com a ordem em que os pares `chave-valor` são armazenados. Ele só registra as conexões entre cada `chave` individual e seu `valor`.

8.9.2 Percorrendo todas as `chaves` de um dicionário com um laço

O método `key()` é conveniente quando não precisamos trabalhar com todos os valores de um dicionário.
Exemplo:

```
favorite_languages = {
  'jen': 'python',
  'sarah': 'c',
  'edward': 'ruby',
  'phil': 'python'
}

for name in favorite_languages.keys():
  print(name.title())
```

```
Jen
Sarah
Edward
Phil
```

Extrai todas as `chaves` do dicionário `favorite_languages` e armazena, uma de cada vez, na variável `name`.

Percorrer as `chaves`, na verdade é o comportamento padrão quando percorremos um dicionário com um laço, portanto este código poderia ser escrito:

```
for name in favorite_languages:
```

em vez de:

```
for name in favorite_languages.keys():
```

Por boa prática optamos pelo método `keys()` pois torna o código mais explicito e de fácil leitura.

O médoto `keys()` não serve apenas para laços, ele devolve uma lista de todas as `chaves`. Exemplo:

```
favorite_languages = {
  'jen': 'python',
  'sarah': 'c',
  'edward': 'ruby',
  'phil': 'python'
}

if 'erin' not in favorite_languages.keys():
  print('Erin, please take our poll!')
```

```
Erin, please take our poll!
```

8.9.3 Percorrendo todas as `chaves` de um dicionário em ordem usando um laço

Um dicionário sempre mantém uma conexão clara entre cada `chave` e seu `valor` associado, mas você não obterá os itens de um dicionário em uma ordem previsível. Isso não é um problema, pois, geralmente, queremos apenas obter o `valor` correto associado a cada `chave`.
Uma maneira de fazer os itens serem devolvidos em determinada sequência é ordenadar as `chaves` à medida que são devolvidas no laço `for`. Podemos usar a função `sorted()` para obter uma cópia ordenada das chaves.
Exemplo:

```
favorite_languages = {
  'jen': 'python',
  'sarah': 'c',
  'edward': 'ruby',
  'phil': 'python'
}

for name in sorted(favorite_languages.keys()):
  print(name.title() +
  ", thank you taking the poll.")
```

```
Edward, thank you taking the poll.
Jen, thank you taking the poll.
Phil, thank you taking the poll.
Sarah, thank you taking the poll.
```

Essa instrução `for` é como as outras instruções `for`, exeto que a função `sorted()` está em torno do método `dictionary.keys()`. Isso diz a Python para listar todas as `chaves` do dicionário e ordernar essa lista antes de percorrê-la com um laço.

8.9.4 Percorrendo todos os `valores` de um dicionário com um laço

Se você tiver mais interessado nos `valores` contidos em um dicionário, o método `values()` pode ser usado para devolver uma lista de `valores` sem as `chaves`.
Exemplo:

```
favorite_languages = {
  'jen': 'python',
  'sarah': 'c',
  'edward': 'ruby',
  'phil': 'python'
}
print("The following languages have been mentioned:")
for language in favorite_languages.values():
  print(language.title())
```

```
The following languages have been mentioned:
Python
C
Ruby
Python
```

A instrução `for`, nesse caso, extrai cada `valor` do dicionário e o armazena na variável `language`. Essa abordagem extrai todos os `valores` do dicionário, sem verificar se há repetições. Isso pode funcionar bem com uma quantidade pequena de `valores`, mas em uma enquete com um número grande de entrevistados, o resultado seria uma lista com muitas repetições. Para ver cada linguagem escolhida sem repetições podemos usar um *conjunto* (`set()`). Um conjunto é semelhante a uma lista exceto que cada item de um conjunto deve ser único.
Exemplo:

```
favorite_languages = {
  'jen': 'python',
  'sarah': 'c',
  'edward': 'ruby',
  'phil': 'python'
}
print("The following languages have been mentioned:")
for language in set(favorite_languages.values()):
  print(language.title())
```

```
The following languages have been mentioned:
```

```
Ruby
Python
C
```

Quando colocamos `set()` em torno de uma lista que contenha itens duplicados, Python identifica os itens únicos na lista e cria um conjunto a partir desses itens. Usamos `set()` para extrair as linguagens únicas em `favorite_languagens.values()`.
O resultado é uma lista de linguagens mencionadaspelas pessoas que participaram da enquete, sem repetições.

8.10 Informações aninhadas

As vezes você vai querer armazenar um conjunto de dicionários em uma lista, uma lista de itens com um valor em um dicionário. Isso é conhecido como **aninhar** informações. podemos aninhar um conjunto de dicionários em uma lista, uma lista de itens em um dicionário ou até mesmo um dicionário em outro dicionário.

8.10.1 Uma lista de dicionários

É comum armazenar vários dicionários em uma lista quando cada dicionário tiver diversos tipos de informação sobre um o mesmo objeto. Todos os dicionários de uma lista devem ter uma estrutura idêntica para que possamos percorrer a lista com um laço e trabalhar com cada objeto representado por um dicionário do mesmo modo.
Exemplo:

```
alien_0 = {'color':'green','points':5}
alien_1 = {'color':'yellow','points':10}
alien_2 = {'color':'red','points':15}

aliens = [alien_0,alien_1,alien_2]

for alien in aliens:
    print(alien)
```

```
{'color': 'green', 'points': 5}
{'color': 'yellow', 'points': 10}
{'color': 'red', 'points': 15}
```

Inicialmente criamos três dicionários, cada um representando um alienígena diferente. Reunimos esse dicionários em uma lista chamada `aliens`. Por fim, percorremos a lista com um laço e exibimos cada alien.

8.10.2 Uma lista em um dicionário

Em vez de colocar um dicionário em uma lista, as vezes é conveniente colocar uma lista em um dicionário. Com uma lista armazenada em um dicionário a lista pode ser apenas um dos aspectos do objeto que estamos descrevendo.
Exemplo:

```
#Armazena informações sobre uma pizza que está sendo pedida
pizza = {'crust':'thick',
          'toppings':['mushrooms','extra cheese']}

#Resumo do pedido
print("You ordered a " +
      pizza['crust'] +
      '-crust pizza ' +
      "with the following toppings:")
for topping in pizza['toppings']:
  print('\t'+topping)
```

```
You ordered a thick-crust pizza with the following toppings:
    mushrooms
    extra cheese
```

Começamos com um dicionário que armazena informações sobre uma pizza que esta sendo pedida. Uma das `chaves` do dicionário é 'crust', e o `valor` associado é a string 'thick'. A próxima `chave`, 'toppings', tem como `valor` uma lista que armazena todos os ingredientes solicitados. Resumimos o pedido antes de preparar a pizza. Para exibir os ingredientes, escrevemos um laço `for`. Para acessar a lista dos ingredientes, usamos a `chave` 'toppings', e o Python obtém a lista de ingredientes do dicionário.

Podemos aninhar uma lista em um dicionário sempre que quisermos que mais de um `valor` seja associado a uma única `chave` em um dicionário. No laço `for` do dicionário, usamos outro laço `for` para percorrer a lista.
Exemplo:

```
favorite_languages = {
    'jen':['python','ruby'],
    'sarah':['c'],
    'edward':['ruby','go'],
    'phil':['python','haskell']
    }
for name, languages in favorite_languages.items():
    print("\n" + name.title() +
          "'s favorite languages are:")
    for language in languages:
        print("\t" + language.title())
```

```
Jen's favorite languages are:
    Python
    Ruby

Sarah's favorite languages are:
    C

Edward's favorite languages are:
    Ruby
    Go

Phil's favorite languages are:
    Python
    Haskell
```

Não aninhe listas e dicionários com muitos níveis de profundidade. Se estiver aninhando itens com um nível de profundidade muito maior do que vimos nos exemplos anteriores ou se estiver trabalhando com o código de outra pessoa, e esse código tiver níveis significativos de informações aninhadas, é mais provável que haja uma maneira mais simples de solucionar o problema existente.

8.10.3 Um dicionário em um dicionário

Podemos aninhar um dicionário com outro dicionário, mas o código poderá ficar complicado rapidamente se isso for feito.
Exemplo:

```
#Dicionário de dicionários
users = {
    'aeinstein':{
        'first':'albert',
        'last':'einstein',
        'location':'princeton'},
    'mcurie':{
        'first':'marie',
        'last':'curie',
        'location':'paris'}
    }

#Extrair informações de um dicionário de dicionários
for username, user_info in users.items():
    print("\nUsername: " + username)
    full_name = user_info['first'] + " " + user_info['last']
    location = user_info['location']
    print("Full name: " + full_name.title())
    print("Location: " + location.title())
```

```
Username: aeinstein
Full name: Albert Einstein
Location: Princeton

Username: mcurie
Full name: Marie Curie
Location: Paris
```

Ao percorremos o dicionário `users` com um laço, o Python armazena cada `chave` na variável `username` e o dicionário associado a cada nome de usuário (`valor`) na variável `user_info`. Para acessar as informações contida no dicionário `user_info` (`valor`), usamos os métodos normais de acessar informações de um dicionário, lembrando que ele esta contido na variável `user_info`.

```
location = user_info['location']
```

Observe que a estrutura do dicionário de cada usuário é idêntica. Embora o Python não exija, essa estrutura facilita trabalhar com dicionários aninhados. Se o dicionário de cada pessoa tivesse chaves diferentes, o código no laço `for` seria mais complicado.

9 Entrada de usuário e laços `while`

A maioria dos programas é escrita para resolver o problema de um usuário final. Para isso, geralmente precisamos obter algumas informações do usuário. Aprenderemos a aceitar dados de entrada do usuário para que o programa possa então trabalhar com eles. Para tal fim, usaremos a função `input()`.
O laço `while` do Python mantém o programa executando enquanto determinadas condições permanecem verdadeiras.

9.1 Entrada de usuário - `input()`

9.1.1 Como a função `input()` trabalha

A função `input()` faz uma pausa em seu programa e espera o usuário fornecer um **texto** (*string*). Depois que o Python recebe a entrada do usuário, esse dado é armazenado em uma variável para que possa ser trabalhado pelo programa, de forma conveniente.
Exemplo:

```
message = input("Tell me soomething, and I will repeat it back to you: ")
print(message)
```

A função `input()` aceita um argumento: O `prompt` - ou as instruções - que queremos que exiba ao usuário para que eles saibam o que devem fazer. Nesse exemplo, quando Python executar a primeira linha, o usuário verá o `prompt` "*Tell me soomething, and I will repeat it back to you:* ". O programa espera enquanto o usuário fornece sua resposta e continua depois que ele teclar **ENTER**. A resposta é armazenada na variável `message`.

O programa **Sublime Text** não executa programas que pedem uma entrada ao usuário. Você pode usar o **Sublime Text** para escrever programas que solicitem uma entrada, mas será necessário executar esses programas a partir de um terminal.
Exemplo de comando para executar um programa a partir de um terminal:

```
python3 parrot.py
```

9.1.2 Escrevendo `prompts` claros

Sempre que usar a função `input()`, inclua um `prompt` claro, fácil de compreender, que informe o usuário exatamente que tipo de informação ele deve passar. Qualquer frase que diga aos usuários o que eles devem fornecer será apropriada.
Exemplo:

```
name = input("Please enter your name: ")
print("Hello, " + name.title() + "!")
```

Acrescente um espaço no final de seus `prompts` (depois dos dois-pontos) para separar o `prompt` da resposta do usuário e deixar claro em que lugar o usuário deve fornecer seu texto.
Ás vezes, você vai querer um `prompt` que seja maior que uma linha. Por exemplo, talvez você queira explicar ao usuário por que está pedindo determinada entrada. Você pode armazenar seu `prompt` em uma variável e passá-la para a função `input()`. Isso permite criar seu `prompt` com várias linhas e escrever uma instrução `input()` clara.
Exemplo:

```
prompt = "If you tell us who you are, we can personalize the messages you see."
prompt += "\nWhat is your name? "
name = input(prompt)
print("Hello, "+name.title()+"!")
```

Esse exemplo mostra uma maneira de criar uma *string* multilinha. A primeira linha armazena a parte inicial da mensagem na variável `prompt`. Na segunda linha, o operador `+=` acrescenta a nova *string* no final da *string* que estava armazenada em `prompt`.

9.1.3 Usando `int()` para aceitar entradas numéricas

Se usarmos a função `input()`, o Python interpretará tudo que o usuário fornecer como uma *string*.
Exemplo:

```
age = input("How old are you? ")
age
```

Sabemos que o Python interpretou a entrada como uma string porque o número agora está entre aspas, se tentar usar a entrada como um número, obterá um erro.

Podemos resolver esse problema usando a função `int()`, que diz ao Python para tratar a entrada como um valor numérico. A função `int()` converte a representação em *string* de um número em uma representação numérica.

```
age = input("How old are you? ")
age = int(age)
age
```

Quando usar uma entrada numérica para fazer cálculos e comparações, lembre-se de converter o valor da entrada em uma representação numérica antes.

9.1.4 Aceitando entradas em Python 2.7

Se você usa Python 2.7, utilize a função `raw_input()` quando pedir uma entrada ao usuário. Essa função interpreta todas as entradas como uma *string*, como faz `input()` em Python 3.
Python 2.7 também tem uma função `input()`, mas essa função interpreta a entrada do usuário como código Python e tenta executá-la. No melhor caso, você verá um erro informando que Python não é capaz de compreender a entrada; no pior caso, executará um código que não pretendia executar. Se tiver usando Python 2.7, utilize `raw_input()` no lugar de `input()`.

9.2 Laço `while`

O laço `for` toma uma coleção de itens e executa um bloco de código uma vez para cada item da coleção. Em comparação, o laço `while` executa durante o tempo em que, ou enquanto, uma determinada condição for verdadeira.

9.2.1 Laço `while` em ação

Podemos usar um laço `while` para contar uma série de números. Exemplo:

```
current_number = 1
while current_number <= 5:
    print(current_number)
    current_number += 1
```

```
1
2
3
4
5
```

O laço `while` é configurado para continuar executando enquanto o valor de `current_number` for menor ou igual a 5. O código no laço exibe o valor `current_number` e então é somado 1 a esse valor com `current_number += 1`. (O operador `+=` é um atalho para `current_number = current_number + 1`)
O Python repete o laço enquanto a condição `current_number <= 5` for verdadeira.

Os programas que você usa no dia a dia provavelmente contêm laços `while`. Por exemplo, um jogo precisa de um laço `while` para continuar executando enquanto você quiser jogar, e pode parar de executar assim que você pedir para sair.

9.2.2 Deixando usuário decidir quando quer sair

Definimos um valor de saída e então deixamos o programa executando enquanto o usuário não tiver fornecido o valor de saída.
Exemplo:

```
prompt = "\nTell me soomething, and I will repeat it back to you:"
prompt += "\n(Enter 'quit' to end the program.)"
prompt += "\n"

message = ""
while message != 'quit':
    message = input(prompt)
    if message != 'quit': #Não repete o 'quit'
        print(message)
```

Primeiro definimos um `prompt` que informa quais são as duas opções ao usuário: fornecer uma mensagem ou o valor de saída (nesse caso, é 'quit'). Em seguida, preparamos uma variável `message` para armazenar o valor que o usuário forneceu. Definimos `message` como uma string vazia, " ", de modo que o Python tenha algo para conferir na primeira vez quealcançar a linha com `while`. Na primeira vez que o programa executar e o Python alcançar a instrução `while`, ele deverá comparar o valor de `message` com 'quit', mas o usuário não forneceu nenhuma entrada.
Se o Python não tiver nada para comparar, ele não será capaz de continar executando o programa. Para resolver esse problema, garantimos que `message` receba algum valor inicial. Embora seja apenas uma *string* vazia, ela fará sentido para o Python e permitirá que a comparação que faz o laço `while` funcionar seja feita. Esse laço `while` executa enquanto o valor de `message` não for 'quit'.
Desde que o usuário não tenha fornecido a palavra 'quit', o `prompt` será exibido novamente e o Python esperará mais entradas. Quando o usuário finalmente digitar 'quit', o Python para de executar o laço `while` e o programa termina.

9.2.3 Usando uma flag

No exemplo anterior, tinhamos um programa que executava determinada tarefa enquanto uma dada condição era verdadeira. E como ficaria em programas mais complicados emque muitos eventos diferentes poderiam fazer o programa parar de executar?
Se muitos eventos possíveis puderem ocorrer para o programa terminar, tentar testar todas essas condições em uma única instrução `while` torna-se complicado e difícil.
Para um programa que deva executar somente enquanto muitas condições forem verdadeiras, podemos definir uma variável que determina se o programa como um todo deve estar ativo. Essa variável, chamada `flag`, atua como um sinal para o programa. Podemos escrever nossos programas de modo que executem enquanto a `flag` estiver definida como `True` e parem de executar quando qualquer um dos vários eventos definir o valor da `flag` como `False`. Como resultado, nossa instrução `while` geral precisa verificar apenas uma condição: se a `flag`, no momento, é `True`. Então todos nossos demais testes (para ver se um evento que deve definir a `flag` como `False` ocorreu) podem estar bem organizados no restante do programa.
Exemplo:

```
prompt = "\nTell me soomething, and I will repeat it back to you:"
prompt += "\n(Enter 'quit' to end the program.)"
prompt += "\n"

active = True

message = ""
while active:
    message = input(prompt)
    if message == 'quit':
        active = False
    else:
        print(message)
```

Definimos a variável `active` como `True` para que o programa comece em um estado ativo. Fazer isso simplifica a instrução `while`, pois nenhuma comparação é feita nessa instrução; a lógica é tratada em outras partes do programa. Enquanto a variável `active` permanecer `True`, o laço continuará a executar.
Se o usuário fornecer 'quit', definimos `active` como `False` e o laço `while` é encerrado. Se o usuário fornecer outro dado que não seja 'quit', exibimos essa entrada como uma mensagem.
Esse programa gera a mesma saída do exemplo anterior, em que havíamos colocado o teste condicional diretamente na instrução `while`. Porém, agora que temos uma `flag` para inidicar se o programa como um todo está em um estado ativo, será mais fácil acrescentar outros testes (por exemplo, instruções `elif`) para eventos que devam fazer `active` se tornar `False`. Isso é útil em programas complicados, como jogos, em que pode haver muitos eventos, e qualquer

um deles poderia fazer o programa parar de executar. Quando um desses eventos fizer a `flag active` se tornar `False`, o laço principal do jogo terminará, uma mensagem de *Game Over* poderia ser exibida e o jogador poderia ter a opção de jogar novamente.

9.2.4 Usando `break` para sair de um laço

Para sair de uma laço `while` de imediato, sem executar qualquer código restante no laço, independente do resultado de qualquer teste condicional, utilize a instrução `break`.
Exemplo:

```
prompt = "\nPlease enter the name of a city you have visited:"
prompt += "\n(Enter 'quit' when you are finished.)"
prompt += "\n"

while True:
    city = input(prompt)

    if city == 'quit':
        break
    else:
        print("I'd love to go to " +
              city.title() + "!")
```

Um laço que comece com `while True` executará indefinidamente, a menos que alcance uma instrução `break`. O laço desse programa continuará pedindo aos usuários para que entrem com os nomes das cidades em que eles estiveram até que 'quit' seja fornecido. Quando 'quit' for digitado, a instrução `break` é executada, fazendo o Python sair do laço.

Você pode usar a instrução `break` em qualquer laço do Python. Por exemplo, `break` pode ser usado para sair de um laço `for` que esteja percorrendo uma lista ou um dicionário.

9.2.5 Usando `continue` em um laço

Em vez de sair totalmente de um laço sem executar o restante de seu código podemos usar a instrução `continue` para retornar ao início, com base no resultado de um teste condicional. Exemplo:

```
current_number = 0
while current_number < 10:
    current_number += 1
    if (current_number % 2) == 0: #Se for par
        continue
    print(current_number)
```

```
1
3
5
7
9
```

No laço conta de 1 a 10, mas apresenta apenas os números impares desse intervalo.
Se o módulo for 0 (o que significa que current_number é divisível por 2), a instrução `continue` diz ao Python para ignorar o restante do laço e voltar ao início. Se o número atual não for divisível por 2, o restante do laço será executado e o Python exibirá o número atual.

9.2.6 Evitando loops infinitos

Todo laço `while` precisa de uma maneira de interromper a execução para que não continue executando indefinidademente.
Exemplo:

```
x = 1
while x <= 5 :
    print(x)
    #x += 1
```

Se for omitida a linha `x += 1` por acidente, o laço executará para sempre.
Agora o valor de x começará em 1, mas jamais será modificado. Como resultado, o teste condicional `x <= 5` será sempre avaliado como `True` e o laço `while` executará indefinidamente, exibindo uma série de 1s.

Todo programador escreve ocasionalmente um loop infinito (ou laço infinito) com `while` por acidente, em especial quando os laços do programa tiverem condições de saída sutis. Se o programa ficar preso em um loop infinito, tecle `CTRL-C` ou simplesmente feche a janela do terminal que está exibindo a saída de seu programa.
Para evitar escrever loops infinitos, teste todos os laços `while` e certifique-se de que eles serão encerrados conforme esperado. Se quiser que seu programa termine quando o usuário fornecer determinado valor de entrada, analise cuidadosamente o modo como seu tratará o valor que deveria fazer o laço parar. Garanta que pelo menos uma parte do programa possa fazer a condição do laço ser `False` ou fazer uma instrução `break` ser alcançada.

Alguns editores, como o Sublime Text, tem uma janela de saída incluida. Isso pode dificultar a interrupção de um loop infinito, e talvez seja necessário fechar o editor para encerrar o laço.

9.3 Usando um laço `while` com listas e dicionários

Para controlar muitos usuários e informações, precisamos usar listas e dicionários com os laço `while`.
Um laço `for` é eficiente para percorrer uma lista, mas você não deve modificiar uma lista em um laço `for`, pois o Python terá problemas para manter o controle dos itens da lista. Para modificar uma lista enquanto trabalha com ela, utilize um laço `while`. Usar laços `while` com listas e dicionários permite coletar, armazenar e organizar muitas entradas a fim de analisá-las e apresentá-las posteriormente.

9.3.1 Transferindo itens de uma lista para outra

Uma maneira de transferir itens de uma lista para outra lista seria usar um laço `while`, a medida que os dados são trabalhados são transferidos de uma lista para outra.
Exemplo:

```
#Transferindo itens de uma lista para outra, usando while

unconfirmed_users = ["alice","brian","candace"]
confirmed_users = []

while unconfirmed_users: #O laço continuar enquanto a lista não for vazia
    current_user = unconfirmed_users.pop() #pesca o ultimo item da lista

    print("Verifying user: " + current_user.title())
    confirmed_users.append(current_user) #Adiciona o item na lista
print("\nThe following users have been confirmed:")
for confirmed in confirmed_users:
    print(confirmed.title())
```

```
Verifying user: Candace
Verifying user: Brian
Verifying user: Alice

The following users have been confirmed:
Candace
Brian
Alice
```

O laço `while` é executado enquanto a lista `unconfirmed_users` não tiver vazia. Nesse laço, o método `.pop()` remove os usuários não verificados, um de cada vez, do final de `unconfirmed_users`. Nesse caso, como Candace é o último elemento da lista

`unconfirmed_users`, seu nome será o primeiro a ser removido, armazenado em `current_user` e adicionado a lista `confirmed_users`. O próximo é Brian e, depois, Alice.

9.3.2 Removendo todas as instâncias de valores específicos de uma lista

Usamos `remove()` para remover umvalor específico de uma lista. A função `remove()` era apropriada porque o valor em que estávamos interessados aparecia apenas uma vez na lista. Porém, e se quiséssemos remover da lista todas as instâncias de um valor?
Suponha que tenhamos uma lista de animais de estimação com o valor 'cat' repetido várias vezes, e desejamos remover todos os 'cat'.
Exemplo:

```
#Removendo todas as instâncias de valores especificos de uma lista

pets = ["dog","cat","dog","goldfish","cat","rabbit","cat"] #Lista
print(pets)

while 'cat' in pets: #Enquanto 'cat' contido na lista pets faça:
    pets.remove('cat') #Remove o primeiro 'cat' que aparecer, a cada iteração

print(pets) #Nova lista
```

```
['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']
['dog', 'dog', 'goldfish', 'rabbit']
```

Começamos com uma lista contendo várias instâncias de 'cat'. Após exibir a lista, o Python entra no laço `while`, pois encontra o valor 'cat' na lista pelo menos uma vez. Depois que entrar no laço, o Python remove a primeira instância de 'cat', retorna para a linha `while` e então entra novamente no laço quando descobre que 'cat' ainda existe na lista. Cada instância de 'cat' é removida até que o valor não esteja mais na lista; nesse momento, o Python sai do laço e exibe a lista novamente.

9.3.3 Preenchendo um dicionário com dados de entrada do usuário

Podemos pedir a quantidade de entrada que for necessária a cada passagem por um laço `while`. Armazenamos os dados coletados em um dicionário.
Exemplo:

```
#Preenchendo um dicionário com dados de entrada e saída

#Inicializando um dicionário vazio
responses = {}

#Flag
polling_active = True

while polling_active:
    #Pede o nome da pessoa e a resposta
    name = input("\nWhat's your name? ")
    response = input("Which mountain would you to climb someday? ")

    #Armazenando resposta no dicionári,o name como chave e response como valor.
    responses[name] = response

    #Criterio de saída do laço, ou nova enquete
    repeat = input("Would you like to let another person respond? (yes/no) ")
    if repeat == 'no':
        polling_active = False

#Percorrendo o dicionário
#Enquete concluída, mostra o resultado
print("\n--- Poll Results ---")
for name, response in responses.items():
    print(name.title() + " would like to climb " + response + ".")
```

O programa inicialmente define um dicionário vazio (`responses`) e cria uma `flag` (`polling_active`) para indicar que a enquete esta ativa. Enquanto `polling_active` for `True`, o Python executará o código que está no laço `while`.
Nesse laço é solicitado ao usuário que entre com seu nome e montanha que gostaria de escalar. Essa informação é armazenada no dicionário `responses` (nome é a `chave` e resposta é `valor` relacionado a `chave`), e uma pergunta é feita ao usuário para saber se ele quer que a enquete continue. Se o usuário responder 'yes', o programa entrará no laço `while` novamente. Se responder 'no', a `flag polling_active` será definida como `False`, o laço `while` para de executar e o último bloco de código exibe o resultado da enquete.

10 Funções

Funções, que são blocos de código nomeados, concebidos para realizar uma tarefa específica. Quando queremos executar uma tarefa em particular, definida em um função, chamamos o nome da função responsável por ela. Se precisar executar essa tarefa várias vezes durante seu programa, não será necessário digitar todo o código para a mesma tarefa repetidamente, basta chamar a função dedicada ao tratamento dessa tarefa e a chamada dirá ao Python para executar o código da função.
sobre as maneiras de passar informações as funções, escrevemos determinadas funções cuja tarefa principal seja exibir informações e outras funções que visam a processar dados e devolver um valor ou um conjunto de valores. Veremos como armazenar funções em arquivos separados, chamados de *módulos*, para ajudar a organizar os arquivos principais de seu programa.

10.1 Definindo uma função

Sintaxe:

```
def nome_funcao (parâmetro_1,...):
    """docstrig"""
    bloco de programação

nome_funcao(argumento_1,...)
```

Exemplo:

```
def greet_user(): #Define o nome e parâmetros da função
    """Exibe um saudação simples.""" #Docstring
    print("Hello world!") #Bloco de programação

greet_user() #Chama a função
```

```
Hello world!
```

Esse exemplo mostra a estrutura mais simples possível para uma função. A primeira linha utiliza a palavra `def` para informar ao Python que estamos definindo uma função. Essa é a definição da função, que informa o nome da função ao Python e, se for aplicável, quais os tipos de informação necessários a função para que ela faça sua tarefa (parâmetros). Os parênteses contêm essa informação. Nesse caso, o nome da função é `greet_user()`, e ela não precisa de nenhuma informação para executar sua tarefa, portanto os parênteses estão vazios. (Mesmo assim, eles são obrigatórios.) Por fim, a definição termina com dois-pontos.

Qualquer linha indentada após `def greet_user():` faz parte do corpo da função. O texto na segunda linha é um comentário chamado `docstring`, que descreve o que a função faz. As `docstring` são colocadas entre aspas triplas, que o Python procura quando gera a documentação das funções de seus programas.
Quando quiser usar essa função, você deve chamá-la. Uma chamada de função diz ao Python para executar o código da função. Para chamar a função, escreva o nome dela, seguido de qualquer informação necessária entre parênteses (argumentos, se for o caso).

10.1.1 Passando informação para uma função

Muitas vezes para que a função faça algo é necessário que especifique um parâmetro, entre os parênteses, da definição da função em `def greet_users()`.
Ao definir um parâmetro, permitimos que a função aceite qualquer valor que você especificar como parâmetro. A função agora espera que um valor seja fornecido para variável parametro sempre que ela for chamada. Ao chamar `greet_user()`, você poderá lhe passar um nome, por exemplo, 'jesse', entre parênteses.
Exemplo:

```
def greet_user(username): #Define o nome e parâmetros da função
    """Exibe um saudação simples.""" #Docstring
    print("Hello " + username.title() + "!") #Bloco de programação

greet_user('jesse') #Chama a função e passa um argumento
```

```
Hello Jesse!
```

10.1.2 Argumentos e parâmetros

A variável `username` na definição da função `greet_user()` (exemplo anterior) é um exemplo de *parâmetro*, uma informação de que a função precisa para executar sua tarefa. O valor 'jesse' em `greet_user('jesse')` é um exemplo de *argumento*. Um argumento é uma informação passada para uma função em sua chamada. Quando chamamos a função, colocamos entre parênteses o valor com que queremos que a função trabalhe. Nesse caso, o argumento 'jesse' foi passado para a função `greet_user()` e o valor foi armazenado no parâmetro `username`.
As vezes, as pessoas falam de argumentos e parâmetros de modo indistinto. Não fique surpreso se vir as variáveis de uma definição de função serem referenciadas como argumentos, ou as variáveis de uma chamada de função serem chamadas de parâmetros.

10.2 Passando argumentos

Pelo fato de ser possível que uma definição de função tenha vários parâmetros, uma chamada de função pode precisar de diversos argumentos. Os argumentos podem ser passados para as funções de várias maneiras. Podemos usar *argumentos posicionais*, que devem estar na mesma ordem em que os parâmetros foram escritos, *argumentos nomeados* (keyword arguments), em que cada argumento é constituído de um nome de variável e de um valor, ou por meio de listas e dicionários de valores.

10.2.1 Argumentos posicionais

Quando chamamos uma função, o Python precisa fazer a correspondência entre cada argumento da chamada da função e um parâmetro da definição. A maneira mais simples de fazer isso é contar com a ordem dos argumentos fornecidos. Valores cuja correspondência seja feita dessa maneira são chamados de *argumentos posicionais*.
Exemplo:

```
def describe_pet(animal_type, pet_name):
    """Exibe informações sobre um animal de estimação."""
    print("\nI have a " + animal_type + ".")
    print("My " + animal_type + "'s name is " + pet_name.title() + ".")

describe_pet('hamster', "harry") #Argumento posicional
```

```
I have a hamster.
My hamster's name is Harry.
```

A definição mostra que essa função precisa de um tipo de animal e de seu nome. Quando chamamos `describe_pet()`, devemos fornecer o tipo de animal e um nome, nessa ordem. Por exemplo, na chamada da função, o argumento 'hamster' é armazenada no parâmetro `animal_type` e o argumento 'harry' é armazenado no parâmetro `pet_name`.

10.2.2 Várias chamadas de função

Podemos chamar uma função quantas vezes forem necessárias.
Exemplo:

```
def describe_pet(animal_type, pet_name):
    """Exibe informações sobre um animal de estimação."""
    print("\nI have a " + animal_type + ".")
    print("My " + animal_type + "'s name is " + pet_name.title() + ".")

describe_pet('hamster', "harry") #Argumento posicional
describe_pet('dog', "willie") #Várias chamadas de função
```

```
I have a hamster.
My hamster's name is Harry.

I have a dog.
My dog's name is Willie.
```

Nessa segunda chamada da função, passamos os argumentos 'dog' e 'willie' a `describe_pet()`. Assim como no conjunto anterior de argumentos que usamos, o Python faz a correspondência entre 'dog' e o parâmetro `animal_type` e entre 'willie' e o parâmetro `pet_name`. Como antes, a função faz sua tarefa, porém, dessa vez, exibe valores para um cachorro chamado Willie. Agora temos um hamster chamado Herry e um cachorro chamado Willie.
Chamar uma função várias vezes é uma maneira eficiente de trabalhar. O código que descreve um animal de estimação é escrito uma só vez na função. Então, sempre que quiser descrever um novo animal de estimação, podemos chamar a função com as informações sobre o animal. Mesmo que o código para descrever um animal de estimação fosse expandido atingindo dez linhas, poderíamos ainda descrever um novo animal de estimação chamando a função novamente com apenas uma linha.
Podemos usar tantos argumentos pasicionais quantos forem necessários nas funções. O Python trabalha com os argumentos fornecidos na chamada da função e faz a correspondência de cada um com o parâmetro associado na definição da função.

10.2.3 A ordem é importante em argumentos posicionais

Podemos obter resultados inesperados se confundirmos a ordem dos argumentos em uma chamada de função quando argumentos posicionais forem usados.
Exemplo:

```
def describe_pet(animal_type, pet_name):
    """Exibe informações sobre um animal de estimação."""
    print("\nI have a " + animal_type + ".")
    print("My " + animal_type + "'s name is " + pet_name.title() + ".")

describe_pet('harry', 'hamster') #Argumento posicional errado
```

```
I have a harry.
My harry's name is Hamster.
```

Se obtivermos resutados engraçados, verifique se a ordem dos argumentos em sua chamada de função corresponde à ordem dos parâmetros na definição da função.

10.2.4 Argumentos nomeados

Um *argumento nomeado* (keyword argument) é um par nome-valor passado para uma função. Associamos diretamente o nome e o valor no próprio argumento para que não haja confusão quando ele for passado para a função. Argumentos nomeados fazem com que você não precise se preocupar com a ordem correta de seus argumentos na chamada da função e deixam claro o papel de cada valor na chamada.
Exemplo:

```
def describe_pet(animal_type, pet_name):
    """Exibe informações sobre um animal de estimação."""
    print("\nI have a " + animal_type + ".")
    print("My " + animal_type + "'s name is " + pet_name.title() + ".")

describe_pet(animal_type="hamster", pet_name="harry") #Argumentos nomeados
```

```
I have a hamster.
My hamster's name is Harry.
```

Quando chamamos a função, dizemos explicitamente ao Python a qual parâmetro cada argumento deve corresponder. Quando o Python lê a chamada da função, ele sabe que deve armazenar o argumento 'hamster' no parâmetro `animal_type` e o argumento 'harry' em `pet_name`.
A ordem dos argumentos nomeados não importa, pois o Python sabe o que é cada valor.

Quando usar argumentos nomeados, lembre-se de usar os nomes exatos dos parâmetros usados na definição da função.

10.2.5 Valores default

Ao escrever uma função, podemos definir um valor *default* para cada parâmetro. Se um argumento para um parâmetro for especificado na chamada da função, o Python usará o valor desse argumento. Se não for, o valor *default* do parâmetro será utilizado. Portanto, se um valor *default* for definido para um parâmetro, você poderá excluir o argumento correspondente, que normalmente seria especificado na chamada da função. Usar valores *default* pode simplificar suas chamadas de função e deixar mais claro o modo como suas funções normalmente são utilizadas.
Exemplo:

```
def describe_pet(pet_name,animal_type='dog'): #Definindo default
    """Exibe informações sobre um animal de estimação."""
    print("\nI have a " + animal_type + ".")
    print("My " + animal_type + "'s name is " + pet_name.title() + ".")

describe_pet(pet_name="willie")
```

```
I have a dog.
My dog's name is Willie.
```

Mudamos a definição de `describe_pet()` para incluir um valor default igual a 'dog' para `animal_type`. A partir de agora, quando a função for chamada sem um `animal_type` especificado, o Python saberá que deve usar o valor 'dog' para esse parâmetro.
Observe que a ordem dos parâmetros na definição da função precisou ser alterada. Como o uso do valor default faz com que não seja necessário especificar o tipo de animal como argumento, o único argumento restante na chamada da função é o nome do animal de estimação. O Python continua interpretando esse valor como um argumento posicional, portanto, se a função for chamda somente com o nome do animal de estimação, esse argumento corresponderá ao primeiro parâmetro listado na definição da função. Esse é o motivo pelo qual o primeiro parâmetro deve ser `pet_name`.

Quando um argumento explicito para `animal_type` for especificado, o Python ignorará o valor default do parâmetro.

Ao usar valores default, qualquer parâmetro com um valor desse tipo deverá ser listado após todos os parâmetros que não tenham valores default. Isso permite que o Python continue a interpretar os argumentos posicionais corretamente.

10.2.6 Chamadas de função equivalente

Como os argumentos posicionais, os argumentos nomeados e os valores default podem ser usados em conjunto, e com frequência você terá várias maneiras equivalentes de chamar uma função.

```
def describe_pet(pet_name,animal_type='dog'):
```

Com essa definição, um argumento sempre deverá ser fornecido para `pet_name` e esse valor pode ser especificado por meio do formato posicional ou nomeado.
Todas as chamadas a seguir serão adequadas a essa função:

```
#Um cachorro chamado willie
describe_pet('willie')
describe_pet(pet_name='willie')

#Um hamster chamado harry
describe_pet('harry','hamster')
describe_pet(pet_name='harry',animal_type='hamster')
describe_pet(animal_type='hamster',pet_name='harry')
```

Cada uma dessas chamadas de função produzirá a mesma saída.

O estilo de chamada que você usar relamente não importa. Desde que suas chamadas de função gerem a saída desejada, basta usar o estilo que achar mais fácil de entender.

10.2.7 Evitando erros em argumentos

Quando começar a usar funções, não se surpreenda se você se deparar com erros sobre argumentos sem correspondência. Argumentos sem correspondência ocorrem quando fornecemos menos ou mais argumentos necessários à função para que ela realize sua tarefa.

10.3 Valores de retorno

Uma função nem sempre precisa exibir sua saída diretamente. Em vez disso, ela pode processar alguns dados e então devolver um valor ou um conjunto de valores. O valor devolvido pela função é chamado de *valor de retorno*. A instrução `return` toma um valor que esta em uma função e o envia de volta a linha que a chamou. Valores de retorno permitem passar boa parte do trabalho pesado de um programa para funções, o que pode simplificar o corpo do programa.

10.3.1 Devolvendo um valor simples

Exemplo:

```
def get_formatted_name(first_name,last_name):
    """Devolve um nome completo formatado de modo elegante."""
    full_name = first_name + " " + last_name
    return full_name.title() #Retorna valor da função

musician = get_formatted_name('jimi', 'hendrix')
print(musician)
```

```
Jimi Hendrix
```

A definição `get_formatted_name()` aceita um primeiro nome e um sobrenome como parâmetros. A função combina esses dois nomes, acrescenta um espaço entre eles e armazena o resultado na variável `full_name`. O valor `full_name` é convertido para que tenha letras iniciais maiúsculas e é devolvido para a linha que fez a chamada.
Quando chamamos uma função que devolve um valor, precisamos fornecer uma variável em que o valor de retorno possa ser armazenada. Nesse caso, o valor devolvido é armazenado na variável `musician`.

10.3.2 Deixando um argumento opcional

As vezes faz sentido criar um argumento opcional para que as pessoas que usarem a função possam optar por fornecer informações extras somente se quiserem. Valores default podem ser usados para deixar um argumento opcional.
Exemplo:

```
def get_formatted_name(first_name,last_name,middle_name=''): #add parâmetro opcional
    """Devolve um nome completo formatado de modo elegante."""
    if middle_name: #middle_name vazio é false (else), e com algo é true (if)
        full_name = first_name + " " + middle_name + " " + last_name
    else:
        full_name = first_name + " " + last_name
    return full_name.title() #Retorna valor da função

musician = get_formatted_name('john', 'hooker','lee')
print(musician)
```

```
John Lee Hooker
```

Os nomes do meio nem sempre são necessários. Para deixar o nome do meio opcional, podemos associar um valor default vazio ao argumento `middle_name` e ignorá-lo, a menos que o usuário forneça um valor. Para que `get_formatted_name()` funcione sem um nome do meio, definimos o valor default de `middle_name` como uma string vazia e o passamos para o final da lista de parâmetros.
Nesse exemplo, o nome é criado a partir de três partes possíveis. Como o primeiro nome e o sobrenome sempre existem, esses parâmetros são listados antes na definição da função. O nome do meio é opcional, portanto é listado por último na definição, e o seu valor default é uma string vazia.
No corpo da função verificamos se o nome do meio foi especificado. O Python interpreta strings não vazias como `True`, portanto `if middle_name` será avaliado como `True` se o argumento para o nome do meio estiver na chamada da função. Se um nome do meio for especificado, o primeiro nome, o nome do meio, e o sobrenome serão combinados para compor um nome completo. Esse nome é então alterado para que as iniciais sejam maiúsculas e é devolvido para linha que chamou a função: ele será armazenado em uma variável `musician` e exibido. Se o nome do meio não for especificado, a string vazia falhará no teste `if` e o bloco `else` será executado. O nome completo será composto apenas do primeiro nome e o sobrenome, e o nome formatado é devolvido para a linha que fez a chamada: ele será armazenado em `musician` e exibido.
Valores opcionais permitem que as funções tratem uma grande variedade de casos de uso, ao mesmo tempo que simplifica ao maxímo as chamadas de função.

10.3.3 Devolvendo um dicionário

Uma função pode devolver qualquer tipo de valor necessário, incluindo estruturas de dados mais complexas como listas e dicionários.
Exemplo:

```
def build_person(first_name,last_name,age=''):
    """Devolve um dicionário com informações sobre uma pessoa."""
    person = {'first':first_name,'last':last_name}
    if age: #Se age não vazio, então True, logo adiciona ao dicionário
        person['age']=age
    return person

musician = build_person('jimi', 'hendrix',age=27)
print(musician)
```

```
{'first': 'jimi', 'last': 'hendrix', 'age': 27}
```

A função `build_person()` aceita um primeiro nome e um sobrenome e então reúne esses valores num dicionário. O valor `first_name` é armazenado com a chave `first` e o valor de `last_name` é armazenado com a chave `last`. O valor de retorno com duas informações textuais originais agora armazenadas em um dicionário.
Essa função aceita informações textuais simples e as coloca em uma estrutura de dados mais significativa, que permite trabalhar com as informações além de simplesmente exibi-las.
É adicionado um novo parâmetro opcional `age` a definição de função e atribuido um valor default vazio ao parâmetro. Se a chamada da função incluir um valor para esse parâmetro, ele será armazenado no dicionário. Essa função sempre armazena o nome de uma pessoa, mas também pode ser modificada para guardar outras informações que você quiser sobre ela.

10.3.4 Usando uma função com um laço `while`

Podemos usar funções com todas as estruturas Python que conhecemos até agora. Por exemplo, vamos usar a funçao `get_formatted_name()` com um laço `while` para saudar os usuários de modo mais formal.

```
def get_formatted_name(first_name,last_name):
    """Devolve um nome completo formatado de modo elegante."""
    full_name = first_name + " " + last_name
    return full_name.title()

while True:
    print("\nPlaese tell me your name? ")
    print("(Enter 'q' at any time to quit)")
    f_name = input("First name: ")
    if f_name == "q":
        break
    l_name = input("Last name: ")
    if l_name == "q":
        break
    formatted_name = get_formatted_name(f_name, l_name)
    print("\nHello, " + formatted_name + "!")
```

Adicionamos uma mensagem que informa como o usuário pode sair e então encerramos o laço se o usuário fornecer o valor de saída em qualquer um dos `prompts`. Agora o programa continuará saudando as pessoas até que alguém forneça 'q' em algum dos nomes.

10.4 Passando uma lista para uma função

Com frequência, você achará útil passar uma lista para uma função, seja uma lista de nomes, de números ou de objetos mais complexos, como dicionários. Se passarmos uma lista a uma função, ela terá acesso direto ao conteúdo dessa lista.
Exemplo:

```
def greet_users(names):
    """Exibe uma saudação simples a cada usuário da lista."""
    for name in names:
        print("Hello, " + name.title() + "!")

usernames = ['hannah', 'ty', 'margot']
greet_users(usernames)
```

```
Hello, Hannah!
Hello, Ty!
Hello, Margot!
```

Definimos `greet_users()` para que espere uma lista de nomes, que é armazenada no parâmetro `names`. A função percorre a lista recebida com um laço e exibe uma saudação para cada usuário. Definimos uma lista de usuários e então passamos a lista `usernames` para `greet_users()` em nossa chamada de função.

10.4.1 Modificando uma lista em uma função

Quando passamos uma lista a uma função, ela pode ser modificada. Qualquer alteração feita na lista no corpo da função é permanente, permitindo trabalhar de modo eficiente, mesmo quando lidamos com grandes quantidades de dados.(Ver **escopo de variável**, sub-tópico).

Exemplo de programa sem funções, trabalhando com listas:

```
#Sem função, trabalhando com listas

#Começa com alguns designs que devem ser impressos.
unprinted_designs = ['iphone case','robot pendant','dodecahedron']
completed_models = []

#Simula a impressão de cada design, até que não haja mais nenhum
#Transfere cada design para completed_models após a impressão
while unprinted_designs:
    current_design = unprinted_designs.pop()

    #Simula a criação de uma impressão 3D a partir do design
    print("Printing model: " + current_design)
    completed_models.append(current_design)

#Exibe todos os modelos finalizados
print("\nThe following models have been printed:")
for complet_model in completed_models:
    print(complet_model)
```

```
Printing model: dodecahedron
Printing model: robot pendant
Printing model: iphone case

The following models have been printed:
dodecahedron
robot pendant
iphone case
```

Exemplo de funções trabalhando com listas:

```
def print_models(unprinted_designs, completed_models):
    """
    Simula a impressão de cada design, até que não haja mais nenhum.
    transfere cada design para completed_models após a impressão.
    """
    while unprinted_designs: #Enquanto a lista não for vazia o loop continua
        current_design = unprinted_designs.pop()

        #Simula a criação de uma impressão 3D a partir do design
        print("Printing model: " + current_design)
        completed_models.append(current_design)

def show_completed_models(completed_models):
    """Exibe todos os modelos finalizados"""
    print("\nThe following models have been printed:")
    for complet_model in completed_models: #Percorre a lista
        print(complet_model)

#Alguns designs que devem ser impressos.
unprinted_designs = ['iphone case','robot pendant','dodecahedron']
completed_models = []

print_models(unprinted_designs, completed_models)
show_completed_models(completed_models)
```

```
Printing model: dodecahedron
Printing model: robot pendant
Printing model: iphone case

The following models have been printed:
dodecahedron
robot pendant
iphone case
```

Definimos a função `print_models()` com dois parâmetros: uma lista de designs a serem impressos e uma lista de modelos concluídos. Dadas essas duas listas, a função simula a impressão de cada design esvaziando a lista de designs não impressos e preenchendo a lista de designs completos. Também definimos a função `show_completed_models()` com um parâmetro: a lista de modelos finalizados. Dada essa lista, `show_completed_models()` exibe o nome de cada modelo impresso.

Esse exemplo também mostra a ideia de que toda função deve ter uma tarefa específica. A primeira função imprime cada design, enquanto a segunda mostra os modelos concluídos. Isso é mais vantajoso que usar uma única função para executar as duas tarefas. Se você estiver escrevendo uma função e perceber que ela esta fazendo muitas tarefas diferentes, experimente dividir o código em duas funções. Lembre-se de que você sempre pode chamar uma função a partir de outras funções, o que pode ser conveniente quando dividimos uma tarefa complexa em uma série de passos.

10.4.1.1 Variável

Uma variável é um rótulo ou um nome dado a um determinado local na memória. Esse local contém o valor que você deseja que o programa memorize, para uso posterior. O que é ótimo em Python é que você não precisa declarar explicitamente qual é o tipo de variável que deseja definir.

O Python tem algumas regras que você deve seguir ao criar uma variável:

1. Ele só pode conter letras (maiúsculas ou minúsculas), números ou o caractere de sublinhado "_".
2. Ele pode não começar com um número.
3. Pode não ser uma palavra-chave (você aprenderá sobre elas mais adiante).

10.4.1.2 Escopo de variável

É o limite das variáveis em um programa - seu "escopo". Os quatro escopos diferentes são: **local**, **envolvente**, **global** e **incorporado**.

1. Local:
   Sempre que você definir uma variável em uma função, seu escopo estará somente dentro da função. Ele pode ser acessado do ponto em que é definido até o final da função e existe enquanto a função estiver sendo executada. Isso significa que seu valor não pode ser alterado ou mesmo acessado de fora da função.
2. Global:
   Esse talvez seja o escopo mais fácil de entender. Sempre que uma variável é definida fora de qualquer função, ela se torna uma variável global, e seu escopo é qualquer lugar dentro do programa. Isso significa que ele pode ser usado por qualquer função.

Também podemos associar na variável um parâmetro que muda a propriedade da variável.

1. `global` - torna a variável global:
   `global nome_variável`
2. `local` - torna a variável local:
   `local nome_variável`

10.4.2 Evitando que uma função modifique uma lista

As vezes você vai querer evitar que uma função modifique uma lista. Por exemplo, suponha que você comece com uma lista de designer não impressos e escreva uma função que transfira esses designs para uma lista de modelos terminados, como no exemplo anterior. Talvez você decida que, apesar de ter imprimido todos os designs, vai querer manter a lista original de designs não impressos em seus registros. Porém, como você transferiu todos os nomes de designs de `unprinted_designs`, a lista agora esta vazia, e essa é a única versão da lista que você tem; a lista original se perdeu. Nesse caso, podemos tratar esse problema passando uma cópia da lista para a função, e não a lista original. Qualquer alteração que a função fizer na lista afetará apenas a cópia, deixando a lista original intacta.

Enviando uma cópia de uma lista para uma função:
`nome_da_função(nome_lista[:])`

A notação de fatia [:] cria uma cópia da lista para ser enviada à função.
`print_models(unprinted_designs[:], completed_models)`

A função `printed_models()` pode fazer seu trabalho, pois ela continua recebendo os nomes de todos os designs não impressos. Porém, dessa vez, ela usa uma cópia da lista original de designs não impressos, e não a lista `unprinted_designs` propriamente dita. A lista `completed_models` será preenchida com os nomes dos modelos impressos, como antes, mas a lista original de designs não impressos não será afetada pela função.
Apesa de poder preservar o conteúdo de uma lista passando uma cópia dela para suas funções, você deve passar a lista original para as funções, a menos que tenha um motivo específico para passar uma cópia. Para uma função, é mais eficiente trabalhar com uma lista existente a fim de evitar o uso de **tempo** e de **memória** necessários para criar uma cópia separada, em especíal quando trabalhamos com listas grandes.

10.5 Passando um número arbitrário de argumentos

As vezes não saberá com antecedência quantos argumentos uma função deve aceitar. Felizmente, o Python permite que uma função receba um número arbitrário de argumentos da instrução de chamada.
Exemplo:

```
def make_pizza(*toppings): #Cria uma tupla para receber número aleatório de argumentos
    """Apresenta a pizza que estamos prestes a preparar."""
    print("\nMaking a pizza with the following toppings:")
    for topping in toppings:
        print(topping)

make_pizza('peperoni')
make_pizza('mushrooms','green peppers','extra cheese')
```

```
Making a pizza with the following toppings:
peperoni

Making a pizza with the following toppings:
mushrooms
green peppers
extra cheese
```

No exemplo tem um parâmetro `*toppings`, mas esse parâmetro agrupa tantos argumentos quantos forem fornecidos na linha de chamada.
O asterisco no nome do parâmetro `*toppings` diz ao Python para criar uma **tupla vazia** chamada `toppings` e reunir os valores recebidos nessa tupla. A instrução `print` no corpo da função gera uma saída que mostra que o Python é capaz de tratar uma chamada de função com um valor e outra chamada com três valores. As chamadas são tratadas de modo semelhante. Observe que o Python agrupa os argumentos em uma tupla, mesmo que a função receba apenas um valor.

Essa sintaxe funciona, não importa quantos argumentos a função receba.

Tuplas:
Tuplas são listas em que os itens não são criadas para mudar (listas imutáveis).
Uma tupla se parece com uma lista, exceto por usar parênteses no lugar de colchetes.
Sintaxe:

```
tuplas = (valor_1,valor_2,valor_3,...)
```

10.5.1 Misturando argumentos posicionais e arbitrários

Se quiser que uma função aceite vários tipos de argumentos, o parâmetro que aceita um número arbitrário de argumentos deve ser colocado por último na definição da função. O Python faz a correspondência de argumentos posicionais e nomeados antes, e depois agrupa qualquer argumento remanescente no último parâmetro.
Exemplo:

```
def make_pizza(size,*toppings): #Um argumento posicional e um arbitrário
    #Argumento arbritrário sempre no final
    """Apresenta a pizza que estamos prestes a preparar."""
    print("\nMaking a " + str(size) +
    "-inch pizza with the following toppings:")
    for topping in toppings:
        print(topping)

make_pizza(16,'peperoni') #Chamada com 1 argumento
make_pizza(12,'mushrooms','green peppers','extra cheese')
#Chamada 1 argumento posicional e 3 argumentos arbitrários
```

```
Making a 16-inch pizza with the following toppings:
peperoni

Making a 12-inch pizza with the following toppings:
mushrooms
green peppers
extra cheese
```

Na definição da função, o Python armazena o primeiro valor recebido no parâmetro `size`. Todos os demais valores que vierem depois são armazenados na tupla `toppings`.

10.5.2 Usando argumentos nomeados arbitrários

As vezes você vai querer aceitar um número arbitrário de argumentos, mas não saberá com antecedência qual tipo de informação será passada para a função. Nesse caso, podemos escrever funções que aceitam tantos para `chave-valor` quantos forem fornecidos pela instrução que faz a chamada. Agora ao invés de trabalharmos com **listas** em argumentos arbitrários (argumentos posicionais e arbitrários), trabalhamos com **dicionários** (argumentos nomeados e arbitrários).
Um exemplo envolve criar perfis de usuários: você sabe que obterá informações sobre um usuário, mas não tem certeza quanto ao tipo de informação que receberá. a função `build_profile()` no próximo exemplo sempre aceita um primeiro nome e um sobrenome, mas aceita também um número **arbitrário de argumentos nomeados**.
Exemplo:

```
def build_profile(first,last,**user_info):
    """Constrói um dicionário contendo tudo que sabemos sobre o usuário."""
    profile = {}
    profile['first_name'] = first
    profile['last_name'] = last
    for key, value in user_info.items(): #percorre o dicionário parâmetro
        profile[key] = value #Add keys e value do parâmetro no dicionário criado da função
    return profile #Retorna o dicionário criado

user_profile = build_profile('albert', 'einstein',
                             location = 'princeton',
                             field = 'physics')
print(user_profile)
```

```
{'first_name': 'albert', 'last_name': 'einstein', 'location': 'princeton', 'field': 'physics
```

A definição de `build_profile()` espera um primeiro nome e um sobrenome e permite que o usuário passe tantos pares `nome-valor` quanto ele quiser. Os asteriscos duplos (**) antes do parâmetro `**user_info` fazem o Python criar um dicionário vazio chamado `user_info` e colocar quaisquer `nome-valor` recebidos nesse dicionário. Nessa função, podemos acessar os pares `nome-valor` em `user_info` como fariamos com qualquer dicionário.
No corpo de `build_profile()`, criamos um dicionário vazio chamado `profile` para armazenar o perfil do usuário. Primeiro adicionamos o primeiro nome e o sobre nome nesse dicionário porque sempre receberemos essas duas informações do usuário. Em seguida, percorremos os pares `chave-valor` adicionais do dicionário `user_info` e adicionamos cada par ao dicionário `profile`. Por fim, devolvemos o dicionário `profile` a linha que chamou a função.
O dicionário devolvido contém o primeiro nome e o sobrenome do usuário e, nesse caso, a localidade e o campo de estudo também. A função será apropriada, não importa quantos pares `chave-valor` adicionais sejam fornecidos na chamada da função.

10.6 Armazenando suas funções em módulos

Uma vantagem das funções é a maneira como elas separam blocos de código de seu programa principal. Ao usar nomes descritivos para suas funções, será bem mais fácil entender o seu programa principal. Você pode dar um passo além armazenando suas funções em um arquivo separado chamado **módulo** e, então, *importar* esse módulo em seu programa principal. Uma instrução `import` diz ao Python para deixar o código de um módulo disponível no arquivo de programa em execução no momento.
Armazenar suas funções em um arquivo separado permite ocultar os detalhes do código de seu programa e se concentrar na lógica de nível mais alto. Também permite reutilizar funções em muitos programas diferentes. Quando armazenamos funções em arquivos separados, podemos compartilhar esses arquivos com outros programadores sem a necessidade de compartilhar o programa todo. Saber como importar funções também possibilita usar bibliotecas de funções que outros programadores escreveram.

10.6.1 Importando um módulo completo

Para começar a importar funções, inicialmente precisamos criar um módulo. Um módulo é um arquivo terminado em `.py` que contém o código que queremos importar para o nosso programa. Exemplo:

```
#Criando módulo = um função para importação
def make_pizza(size,*toppings): #parâmetros, um posicional e um arbitrario (tupla)
    """Apresenta a pizza que estamos prestes a preparar."""
    print("\nMaking a " + str(size) +
          "-inch pizza with the following toppings:")
    for topping in toppings:
        print("- " + topping)
```

Agora criamos um arquivo separado chamado `making_pizza.py` no mesmo diretório em que esta `pizza.py`. Este arquivo importa o módulo que acabamos de criar e, em seguida, faz duas chamadas para a função `make_pizza()`.

```
#Importando módulo
import pizza

#Usando função do módulo
#Módulo.função()
pizza.make_pizza(16, 'pepperoni')
pizza.make_pizza(12, 'mushrooms','green peppers','extra cheese')
```

Quando o Python lê esse arquivo, a linha `import pizza` lhe diz para abrir o arquivo `pizza.py` e copiar todas as funções dele para esse programa. Você não vê realmente o código sendo copiado entre os arquivos porque o Python faz isso internamente, a medida que o programa executa. Tudo que você precisa saber é que qualquer função definida em `pizza.py` (arquivo do módulo) agora estará disponível em `making_pizza.py` (arquivo que importa o módulo).
Para chamar uma função que está em um módulo importado, forneça o nome do módulo, que é `pizza` nesse caso, seguido do nome da função, `make_pizza()`, separados por um ponto. Esse código gera a mesma saída que o programa original, que não importava um módulo.

Essa primeira abordagem à importação, em que simplesmente escrevemos `import` seguido do nome do módulo, deixa todas as funções do módulo disponíveis ao seu programa. Se você usar esse tipo de instrução `import` para importar um módulo completo chamado `nome_do_módulo.py`, todas as funções do módulo estarão disponíveis por meio dessa sintaxe a seguir:

```
nome_do_módulo.nome_da_função()
```

10.6.2 Importando funções específicas

Podemos também importar uma função específica de um módulo. Eis a sintaxe geral para essa abordagem:

```
from nome_do_módulo import nome_da_função
```

Você pode importar quantas funções quiser de um módulo separando o nome de cada função com uma virgula:

```
from nome_do_módulo import função_0, função_1, função_2
```

Com essa sintaxe não precisamos usar a notação de ponto ao chamar uma função do módulo. Por exemplo, quando importamos explicitamente a função make_pizza() na instrução `import`, podemos chamá-la pelo nome quando ela for utilizada.

```
#Importando função do módulo explicitamente
from pizza import make_pizza

#Usando função do módulo
#Quando importamos o módulo e explicitamos a função, podemos chamar a função diretamente
#Sem usar: nome_modulo.nome_função()
make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms','green peppers','extra cheese')
```

10.6.3 Usando a palavra reservada `as` para atribuir um alias a uma função

Se o nome de uma função que você importar entrar em conflito com um nome existente em seu programa ou se o nome da função for longo, podemos usar um *alias* único e conciso, que é um nome alternativo, semelhante a um apelido para a função. Dê esse apelido especial a função quando importá-la.
Vamos atribuir um alias `mp()` para a função `make_pizza()` importando `make_pizza as mp`.
A palavra reservada `as` renomeia uma função usando o alias que você fornecer.
Exemplo:

```
#Importando função do módulo explicitamente
#Dando um alias que substitui o nome da função
from pizza import make_pizza as mp

#Usando função do módulo
#Quando importamos o módulo e explicitamos a função, podemos chamar a função diretamente
#Sem explicitar: nome_modulo.nome_função()
mp(16, 'pepperoni')
mp(12, 'mushrooms','green peppers','extra cheese')
```

A instrução `import` no exemplo renomeia a função `make_pizza()` para `mp()` nesse programa. Sempre que quiser chamar `make_pizza()`, você pode simplesmente escrever `mp()` em seu lugar, e o Python executará o código de `make_pizza()`, ao mesmo tempo que evitará confusão com outra função `make_pizza()` que você possa ter escrito nesse arquivo de programa.

A sintaxe geral que fornece um alias é:
`from nome_do_módulo import nome_da_função as nf`

10.6.4 Usando a palavra reservada `as` para atribuir um alias a uma módulo

Também podemos fornecer um alias para um nome de módulo. Dar um alias conciso a um módulo, por exemplo, `p` para `pizza`, permite chamar mais rapidamente as funções do módulo. Chamar `p.make_pizza()` é mais compacto que chamar `pizza.make_pizza()`.
Exemplo:

```
#Importando um módulo
#Dando um alias que substitui o nome do módulo
import pizza as p

#Usando alias do modulo e uma função do módulo
#Usando a módulo: alias_do_modulo.nome_função()
p.make_pizza(16, 'pepperoni')
p.make_pizza(12, 'mushrooms','green peppers','extra cheese')
```

O módulo `pizza` recebe o alias `p` na instrução `import`, mas todas as funções do módulo preservam seus nomes originais. Chamar as funções escrevendo `p.make_pizza()` não só é mais compacto que escrever `pizza.make_pizza()` como também desvia sua atenção do nome do módulo e permite dar enfoque aos nomes descritivos de suas funções. Esses nomes de função, que claramente informam o que cada função faz, são mais importantes para a legibilidade de seu código que usar o nome completo do módulo.

A sintaxe geral para essa abordagem é:
`import nome_do_módulo as nm`

10.6.5 Importando todas as funções de um módulo

Podemos dizer ao Python para importar todas as funções de um módulo usando o operador asterisco (*).
Exemplo:

```
#Importando todas as funções de um módulo
from pizza import *

#Quando importamos o módulo e explicitamos todas as função,
#podemos chamar a função diretamente
make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms','green peppers','extra cheese')
```

O asterisco na instrução `import` diz ao Python para copiar todas as funções do módulo `pizza` para esse arquivo do programa. Como todas as funções são importadas, podemos chamar qualquer função pelo nome, ser usar a notação ponto. No entanto, é melhor não usar essa abordagem quando trabalhar com módulos maiores, que não foram escritos por você: se o módulo tiver um nome de função que seja igual a um nome existente em seu projeto, você poderá ter alguns resultados inesperados. O Python poderá ver várias funções ou variáveis com o mesmo nome e, em vez de importar todasas funções separadamente, ele as sobrescreverá.
A melhor abordagem é importar a função ou as funções que você quiser ou importar o módulo todo e usar a notação de ponto. Isso resulta em um código claro, mais fácil de ler e de entender. Incluí esta seção para que você possa reconhecer instruções `import` como esta quando as vir no código de outras pessoas.

Sintaxe:
`from nome_do_módulo import *`

11 Classes

Na programação orientada a objetos, escrevemos *classes* que representam entidades e situações do mundo real, e criamos *objetos* com base nessas classes. Quando escrevemos uma classe, definimos o comportamento geral que toda uma categoria de objetos pode ter.
Quando criamos objetos individuais a partir da classe, cada objeto será automaticamente equipado com o comportamento geral. Então você poderá dar a cada objeto as características únicas que desejar.
Criar um objeto a partir de uma classe é uma operação conhecida como *instanciação*, e trabalhamos com instâncias de uma classe. Escrevemos classes e criamos instâncias dessas classes. Especificamos o tipo de informação que pode ser armazenada nas instâncias e definimos ações que podem ser executadas nessas instâncias. Também escreveremos classes que estendem a funcionalidade de classes existentes, de modo que classes semelhantes possam compartilhar códigos de forma eficiente. Armazenaremos nossas classes em módulos e importaremos classes escritas por outros programadores para nosso próprios arquivos de programa.

11.1 Criando e usando classes

Podemos modelar de tudo usando classes. Vamos começar escrevendo o exemplo de uma classe simples, `Dog`, que representa um cachorro - não um cachorro em particular, mas qualquer cachorro. O que sabemos sobre a maioria dos cachorros de estimação? Todos eles tem um nome e uma idade. Também sabemos que a maioria deles senta e rola. Essas duas informações (nome e idade) e esses dois comportamentos (senta e rola) farão parte da nossa classe `Dog`, pois são comuns a maioria dos cachorros.
Essa classe dirá ao Python como criar um objeto que represente um cachorro. Depois que nossa classe estiver escrita, ela será usada para criar instâncias individuais, em que cada uma representará um cachorro específico.

11.1.1 Criando uma classe

Cada instância criada a partir da classe `Dog` armazenará um nome (`name`) e uma idade (`age`), e daremos a cada cachorro a capacidade de sentar (`sit()`) e de rolar (`roll_over()`).
Exemplo:

```
#Cria a classe Dog
#Nome da classe em maiusculo por ser uma classe
class Dog():
    """Uma tentativa simples de modelar um cachorro."""

    #Atributos
    def __init__(self, name, age):
        """Inicializando os atributos name e age."""
        self.name = name
        self.age = age

    #Metodos
    def sit(self):
        """Simula um cachorro sentando em resposta a um comando."""
        print(self.name.title() + " is now sitting.")

    def roll_over(self):
        """Simula um cachorro rolando em resposta a um comando."""
        print(self.name.title() + " rolled over!")
```

Definimos uma classe chamada `Dog`. Por convenção, nomes com a primeira letra maiúscula referem-se a classes em Python. Os parênteses na definição da classe estão vazios porque estamos criando essa classe do zero. Também escrevemos um *docstring* que descreve o que essa classe faz.

- Método `__init__()`

  Uma função que faz parte de uma classe é um método. Tudo que aprendemos sobre funções também se aplica aos métodos. A única diferença prática, por enquanto, é o modo como chamamos os métodos. O método `__init__()` é um método especial que no Python executa automaticamente sempre que criamos uma nova instância baseada na classe.
  Esse método tem dois *underscores* no início e dois no final - uma convenção que ajuda a evitar que os nomes default de métodos Python entrem em conflito com nomes de métodos criados por você.

  No exemplo definimos o método `__init__()` para que tenha três parâmetros: `self`, `name` e `age`. O parâmetro `self` é obrigatório na definição do método e deve estar antes dos

demais parâmetros. Deve estar incluído na definição, pois, quando o Python chama esse método `__init__()` depois (para criar uma instância de `Dog`), a chamada do método passará o argumento `self` automaticamente. Toda chamada de método associada a uma classe passa `self`, que é uma referência a própria instância, de modo automático. O parâmetro `self` dá acesso aos atributos e métodos da classe a instância individual. Quando criamos uma instância de `Dog`, o Python chamará o método `__init__()` da classe `Dog`. Passaremos um nome e uma idade como argumentos para `Dog()`, `self` é passado automaticamente, portanto não é preciso especificá-lo. Sempre que quisermos criar uma instância da classe `Dog` forneceremos valores apenas para os dois últimos parâmetros, que são `name` e `age`.

As duas variáveis definidas tem o prefixo `self`. Qualquer variável prefixada com `self` esta disponível a todos os métodos da classe; além disso, podemos acessar essa variável por meio de qualquer instância criada a partir da classe. `self.name = name` usa o valor armazenado no parâmetro `name` e o armazena na variável `name`, que é então associada a instância criada. O mesmo processo ocorre com `self.age = age`. Variáveis como essas, acessíveis por meio de instâncias, são chamadas de *atributos*.

A classe `Dog` tem dois outros dois métodos definidos: `sit()` e `roll_over()`. Como esses dois métodos não precisam de informações adicionais como um nome ou uma idade, simplesmente os definimos com um parâmetro `self`. As instâncias que criarmos posteiormente terão acesso a esses dois métodos. Em outras palavras, elas terão a capacidade de sentar e rolar. Por enquanto, `sit()` e `roll_over()` não fazem muito. Apenas exibem uma mensagem dizendo que o cachorro esta sentando ou rolando. No entanto, o conceito pode ser estendido para situações realistas se essa classe fizesse parte de um jogo de computador de verdade, esse método conteria código para fazer a animação de um cachorro sentando e rolando. Se essa classe tivesse sido escrita para controlar um robô, esses métodos derecionariam os movimentos para fazer um cachorro-robô sentar e rolar.

- Criando classe no Python 2.7

  Quando criar uma classe no Python 2.7, será necessário fazer uma pequena mudança. Inclua o termo `object` entre parênteses quando criá-la:

  ```
  class nomeClasse(object):
  ```

  Exemplo:

  ```
  class Dog(object):
  ```

Observação:
Instanciar um objeto, em Python, significa criar uma instância de uma classe. Quando você cria uma instância de uma classe, você instancia o objeto. No Python, o processo de instanciar objetos envolve criar e inicializar objetos.
Para instanciar uma classe Python, você precisa usar o método construtor, que é o método `__init__()`. O método construtor inicializa os atributos ou propriedades de um objeto.

11.1.2 Criando uma instância a partir de uma classe

Pense em uma classe como um conjunto de instruções para criar uma instância.
Exemplo:

```
#Cria a classe Dog
#Nome da classe em maiusculo por ser uma classe
class Dog():
    """Uma tentativa simples de modelar um cachorro."""

    #Atributos
    def __init__(self, name, age):
        """Inicializando os atributos name e age."""
        self.name = name
        self.age = age

    #Metodos
    def sit(self):
        """Simula um cachorro sentando em resposta a um comando."""
        print(self.name.title() + " is now sitting.")

    def roll_over(self):
        """Simula um cachorro rolando em resposta a um comando."""
        print(self.name.title() + " rolled over!")

my_dog = Dog('willie',6) #Criando uma instância a partir de uma classe

#Acessando atributos
print("My dog's name is " + my_dog.name.title() + ".")
print("My dog is " + str(my_dog.age) + " years old.")
```

```
My dog's name is Willie.
My dog is 6 years old.
```

A classe `Dog` é um conjunto de instruções que diz ao Python como criar instâncias individuais que representam cachorros específicos.

Dizemos ao Python para criar um cachorro de nome 'willie' e idade igual a 6. Quando o Python lê essa linha, ele chama o método `__init__()` de `Dog` com os argumentos 'willie' e 6. O método `__init__()` cria uma instância que representa esse cachorro em particular e define os atributos `name` e `age` com os valores que fornecemos. Esse método não tem uma instrução `return` explicita, mas o Python devolve automaticamente uma instância que representa esse

cachorro. Armazenamos essa instância na variável `my_dog`. A convenção de nomenclatura é útil nesse caso: em geral, podemos supor que um nome com a primeira letra maiúscula como `Dog` refere-se a uma classe, enquanto um nome com letra minúsculas como `my_dog` refere-se a uma única instância criada a partir de uma classe.

11.1.2.1 Acessando atributos

Para acessar os atributos de uma instância utilize a notação de ponto. Acessamos o valor do atributo `name` de `my_dog` escrevendo:
`my_dog.name`

A notação de ponto é usada com frequência em Python. Essa sintaxe mostra como o Python encontra o valor de um tributo. Nesse caso, o interpretador olha para a instância `my_dog` e encontra o atributo `name` associado a ela. É o mesmo atributo refenciado como `self.name` na classe `Dog`. Usamos a mesma arbordagem para trabalhar com o atributo `age`. Em nossa primeira instrução `print`, `my_dog.name.title()` faz com que 'willie' - o valor do atributo `name` de `my_dog` - comece com uma letra maiúscula. Na segunda instrução `print`, `str(my_dog.age)` converta 6 - o valor do atributo `age` de `my_dog` - em uma string.

11.1.2.2 Chamando métodos

Depois que criamos uma instância da classe `Dog`, podemos usar a notação de ponto para chamar qualquer método definido nessa classe.

```
#Cria a classe Dog
#Nome da classe em maiusculo por ser uma classe
class Dog():
    """Uma tentativa simples de modelar um cachorro."""

    #Atributos
    def __init__(self, name, age):
        """Inicializando os atributos name e age."""
        self.name = name
        self.age = age

    #Metodos
    def sit(self):
        """Simula um cachorro sentando em resposta a um comando."""
        print(self.name.title() + " is now sitting.")

    def roll_over(self):
        """Simula um cachorro rolando em resposta a um comando."""
        print(self.name.title() + " rolled over!")

my_dog = Dog('willie',6) #Criando uma instância a partir de uma classe

#Chamando métodos
my_dog.sit()
my_dog.roll_over()
```

```
Willie is now sitting.
Willie rolled over!
```

Para chamar um método, especifique o nome da instância (nesse caso, `my_dog`) e o método que você que chamar, separados por um ponto. Quando o Python lê `my_dog.sit()`, ele procura o método `sit()` na classe `Dog` e executa esse código. A linha `my_dog.roll_over()` é interpretada do mesmo modo.

11.1.2.3 Criando várias instâncias

Você pode criar tantas instâncias de uma classe quantas forem necessárias.

```
#Cria a classe Dog
#Nome da classe em maiusculo por ser uma classe
class Dog():
    """Uma tentativa simples de modelar um cachorro."""

    #Atributos
    def __init__(self, name, age):
        """Inicializando os atributos name e age."""
        self.name = name
        self.age = age

    #Metodos
    def sit(self):
        """Simula um cachorro sentando em resposta a um comando."""
        print(self.name.title() + " is now sitting.")

    def roll_over(self):
        """Simula um cachorro rolando em resposta a um comando."""
        print(self.name.title() + " rolled over!")

#Criando instâncias a partir da classe
my_dog = Dog('willie',6)
your_dog = Dog('lucy', 3)

#Acessando atributos
print("My dog's name is " + my_dog.name.title() + ".")
print("My dog is " + str(my_dog.age) + " years old.")

#Chamando métodos
my_dog.sit()
my_dog.roll_over()

#Acessando atributos
print("\nMy dog's name is " + your_dog.name.title() + ".")
print("My dog is " + str(your_dog.age) + " years old.")

#Chamando métodos
your_dog.sit()
your_dog.roll_over()
```

```
My dog's name is Willie.
My dog is 6 years old.
Willie is now sitting.
Willie rolled over!

My dog's name is Lucy.
My dog is 3 years old.
Lucy is now sitting.
Lucy rolled over!
```

Nesse exemplo, criamos um cachorro chamado ‘willie’ e uma cadela de nome ‘lucy’. Cada cachorro é uma instância separada, com seu próprio conjunto de atributos, capaz de realizar o mesmo conjunto de ações.

Mesmo que usássemos o mesmo nome e a mesma idade para o segundo cachorro, o Python criaria uma instância separada da classe `Dog`. Você pode criar tantas instâncias de uma classe quantas forem necessárias, desde que dê a cada instância um nome de variável único ou que ela ocupe uma única posição em uma lista ou dicionário.

11.2 Trabalhando com classes e instâncias

Podemos usar classes para representar muitas situações do mundo real. Depois que escrever uma classe, você gastará a maior parte do seu tempo trabalhando com instâncias dessa classe. Uma das primeiras tarefas que você vai querer fazer é modificar os atributos associados a uma instância em particular. Podemos modificar os atributos de uma instância diretamente, ou escrever métodos que atualizem os atributos de formas especificas.
Exemplo da classe que iremos trabalhar nesse tópico:

```
#Cria a classe
class Car():
    """Uma tentativa simples de representar um carro."""

    #Inicializa atributos
    def __init__(self, make, model, year):
        """Inicializa os atributos que descrevem um carro."""
        self.make = make
        self.model = model
        self.year = year

    #Métodos
    def get_descriptive(self):
        """Devolve um nome descritivo, formatado de modo elegante."""
        long_name = str(self.year) + " "
        long_name += self.make + " "
        long_name += self.model
        return long_name.title()

#Cria uma instância
new_car = Car('audi', 'a4', 2016)

#Descreve a instância/objeto atraves do método, chama o método
print(new_car.get_descriptive())
```

```
2016 Audi A4
```

11.2.1 Definindo um valor default para um atributo

Todo atributo de uma classe precisa de um valor inicial, mesmo que esse valor seja 0 ou uma string vazia. Em alguns casos, por exemplo, quando definimos um valor default, faz sentido específicar esse valor inicial no corpo do método `__init__()`; se isso for feito para um atributo, você não precisará incluir um parâmetro para ele.

Vamos acrescentar um atributo chamado `odometer_reading` que sempre começa com o valor 0. Também adicionamos um método `read_odometer()` que nos ajudará a ler o hodômetro de cada carro.

```
#Cria a classe
class Car():
    """Uma tentativa simples de representar um carro."""

    #Inicializa atributos
    def __init__(self, make, model, year):
        """Inicializa os atributos que descrevem um carro."""
        self.make = make
        self.model = model
        self.year = year

        #Atributo default
        #Não precisa ser inicializado no parâmetro
        self.odometer_reading = 0 #Atributo com valor default

    #Métodos
    def get_descriptive(self):
        """Devolve um nome descritivo, formatado de modo elegante."""
        long_name = str(self.year) + " "
        long_name += self.make + " "
        long_name += self.model
        return long_name.title()

    def read_odometer(self):
        """Exibe uma frase que mostra a milhagem do carro."""
        print("This car has " + str(self.odometer_reading) + " miles on it.")

#Cria uma instância
new_car = Car('audi', 'a4', 2016)
```

```
#Descreve a instância/objeto atraves do método, chama o método
print(new_car.get_descriptive())
new_car.read_odometer() #Método que imprime atributo default
```

```
2016 Audi A4
This car has 0 miles on it.
```

Dessa vez, quando o Python chama o método `__init__()` para criar uma nova instância, os valores para o fabricante, o modelo e o ano são armazenados como atributos. Em seguida, o Python cria um novo atributo chamado `odometer_reading` e define seu valor inicial com 0. Também temos um novo método de nome `read_odometer()` que facilita a leitura da milhagem de um carro.

11.2.2 Modificando valores de atributos

Você pode alterar o valor de um atributo de três maneiras: podemos modificar o valor diretamente por meio de uma instância, definir o valor com um método ou incrementá-lo (somar um determinado valor a ele). Vamos analisar cada uma dessas abordagens.

11.2.2.1 Modificando o valor de um atributo diretamente

A maneira mais simples de modificar o valor de um atributo é acessá-lo diretamente por meio de uma instância.
Exemplo:

```
#Cria a classe
class Car():
    """Uma tentativa simples de representar um carro."""

    #Inicializa atributos
    def __init__(self, make, model, year):
        """Inicializa os atributos que descrevem um carro."""
        self.make = make
        self.model = model
        self.year = year

        #Atributo default
        #Não precisa ser inicializado no parâmetro
        self.odometer_reading = 0 #Atributo com valor default

    #Métodos
    def get_descriptive(self):
        """Devolve um nome descritivo, formatado de modo elegante."""
        long_name = str(self.year) + " "
        long_name += self.make + " "
        long_name += self.model
        return long_name.title()

    def read_odometer(self):
        """Exibe uma frase que mostra a milhagem do carro."""
        print("This car has " + str(self.odometer_reading) + " miles on it.")

#Cria uma instância
new_car = Car('audi', 'a4', 2016)
```

```
#Descreve a instância/objeto atraves do método, chama o método
print(new_car.get_descriptive())
new_car.read_odometer() #Método que imprime atributo default

#Modificando o valor de um atributo diretamente
new_car.odometer_reading = 23
new_car.read_odometer()
```

```
2016 Audi A4
This car has 0 miles on it.
This car has 23 miles on it.
```

Usamos a notação de ponto para acessar o atributo `odometer_reading` do carro e definir seu valor diretamente. Essa linha diz ao Python para usar a instância `new_car`, encontrar o atributo `odometer_reading` associado a ela e definir o valor atributo com 23.

```
#Modificando o valor de um atributo diretamente
new_car.odometer_reading = 23
```

11.2.2.2 Modificando o valor de um atributo com um método

Pode ser conveniente ter métodos que atualizem determinados atributos para você. Em vez de acessar o atributo de modo direto, passe o novo valor para um método que trate a atualização internamente.
Exemplo:

```
#Cria a classe
class Car():
    """Uma tentativa simples de representar um carro."""

    #Inicializa atributos
    def __init__(self, make, model, year):
        """Inicializa os atributos que descrevem um carro."""
        self.make = make
        self.model = model
        self.year = year

        #Atributo default
        #Não precisa ser inicializado no parâmetro
        self.odometer_reading = 0 #Atributo com valor default

    #Métodos
    ##Descreve o carro atraves dos atributos
    def get_descriptive(self):
        """Devolve um nome descritivo, formatado de modo elegante."""
        long_name = str(self.year) + " "
        long_name += self.make + " "
        long_name += self.model
        return long_name.title()

    ##Imprime odometer na tela
    def read_odometer(self):
        """Exibe uma frase que mostra a milhagem do carro."""
        print("This car has " + str(self.odometer_reading) + " miles on it.")

    ##Método para modificar atributo
    def update_odometer(self,mileage):
        """
        Define o valor de leitura do hodômetro com o valor especificado.
        Rejeita aalteração se for tentativa de definir um valor menor
        para o hodômetro.
        """
```

```
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

#Cria uma instância
new_car = Car('audi', 'a4', 2016)

#Modifica atributo por meio de um método
new_car.update_odometer(23)
new_car.read_odometer()
```

```
This car has 23 miles on it.
```

A única modificação em `Car` foi o acréscimo de `update_odometer()`. Esse novo método aceita um valor de milhagem e o armazena em `self.odometer_reading`. Chamamos `update_odometer()` e lhe passamos o valor 23 como argumento (correspondendo ao parâmetro `mileage` na definição do método). Esse método define o valor de leitura do hodômetro com 23 e `read_odometer()` exibe sua leitura.

`update_odometer()` verifica se o novo valor do hodômetro faz sentido antes de modifcar o atributo. Se a nova milhagem, `mileage`, for maior ou igual a milhagem existente, `self.odometer_reading`, você poderá atualizar o valor de leitura do hodômetro com a nova milhagem. Se a nova milhagem for menor que a existente, você receberá um aviso informando que não pode diminuir o valor lido no hodômetro.

11.2.2.3 Incrementando o valor de um atributo com um método

As vezes, você vai querer incrementar o valor de um atributo de determinada quantidade, em vez de definir um valor totalmente novo.
Exemplo:

```
#Cria a classe
class Car():
    """Uma tentativa simples de representar um carro."""

    #Inicializa atributos
    def __init__(self, make, model, year):
        """Inicializa os atributos que descrevem um carro."""
        self.make = make
        self.model = model
        self.year = year

        #Atributo default
        #Não precisa ser inicializado no parâmetro
        self.odometer_reading = 0 #Atributo com valor default

    #Métodos
    ##Descreve o carro atraves dos atributos
    def get_descriptive(self):
        """Devolve um nome descritivo, formatado de modo elegante."""
        long_name = str(self.year) + " "
        long_name += self.make + " "
        long_name += self.model
        return long_name.title()

    ##Imprime odometer na tela
    def read_odometer(self):
        """Exibe uma frase que mostra a milhagem do carro."""
        print("This car has " + str(self.odometer_reading) + " miles on it.")

    ##Método para modificar atributo
    def update_odometer(self,mileage):
        """
        Define o valor de leitura do hodômetro com o valor especificado.
        Rejeita aalteração se for tentativa de definir um valor menor para o hodômetro.
        """
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
```

```
        else:
            print("You can't roll back an odometer!")

    ##Método para incrementar atributo
    def increment_odometer(self, miles):
        """Soma a quantidade especificada ao valor de leitura do hodômetro."""
        if miles >= 0:
            self.odometer_reading += miles
        else:
            print("You can't roll back an odometer!")

#Cria uma instância
used_car = Car('subaru', 'outback', 2013)

#Modifica atributo por meio de um método
used_car.update_odometer(23500)
used_car.read_odometer()

#Método incrementa o valor de um atributo
used_car.increment_odometer(100)
used_car.read_odometer()
```

```
This car has 23500 miles on it.
This car has 23600 miles on it.
```

O novo método `increment_odometer()` aceita uma quantidade de milhas e soma esse valor a `self.odometer_reading`. Criamos um carro usado, `used_car`. Definimos seu hodômetro com o valor 23.500 chamando `update_odometer()` e passando-lhe o valor 23500. Chamamos `increment_odometer()` e passamos o valor 100 para somar as cem milhas que dirigimos entre comprar o carro e registrá-lo.

11.3 Herança

Nem sempre você precisará começar do zero para escrever uma classe. Se a classe que você estiver escrevendo for uma versão especializada de outra classe já criada, a *herança* poderá ser usada. Quando uma classe herda de outra, ela assumirá automaticamente todos os atributos e métodos da primeira classe. A classe original se chama *classe-pai* e a nova classe é a *classe-filha*. A classe-filha herda todos os atributos e métodos da sua classe-pai, mas também é livre para definir novos atributos e métodos próprios.

11.3.1 Método `__init__()` de uma classe-filha

A primeira tarefa do Python ao criar uma instância de uma classe-filha é atribuir valores a todos os atributos da classe-pai. Para isso, o método `__init__()` de uma classe-filha precisa de ajuda de sua classe-pai.
Exemplo:

```
#Herança

#classe-pai
class Car():
    """Uma tentativa simples de representar um carro."""

    #Inicializa atributos
    def __init__(self, make, model, year):
        """Inicializa os atributos que descrevem um carro."""
        self.make = make
        self.model = model
        self.year = year

        #Atributo default
        #Não precisa ser inicializado no parâmetro
        self.odometer_reading = 0 #Atributo com valor default

    #Métodos
    ##Descreve o carro atraves dos atributos
    def get_descriptive(self):
        """Devolve um nome descritivo, formatado de modo elegante."""
        long_name = str(self.year) + " "
        long_name += self.make + " "
        long_name += self.model
        return long_name.title()
```

```
    ##Imprime odometer na tela
    def read_odometer(self):
        """Exibe uma frase que mostra a milhagem do carro."""
        print("This car has " + str(self.odometer_reading) + " miles on it.")

    ##Método para modificar atributo
    def update_odometer(self,mileage):
        """
        Define o valor de leitura do hodômetro com o valor especificado.
        Rejeita aalteração se for tentativa de definir um valor menor para o hodômetro.
        """
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    ##Método para incrementar atributo
    def increment_odometer(self, miles):
        """Soma a quantidade especificada ao valor de leitura do hodômetro."""
        if miles >= 0:
            self.odometer_reading += miles
        else:
            print("You can't roll back an odometer!")

#Classe-filha
class ElectricCar(Car): #herança da classe-pai 'Car'
    """Representa aspectos de um carro especificos de veiculos elétricos."""

    #Atributos herdados
    def __init__(self, make, model, year):
        """Inicializa os atributos da classe-pai."""
        super().__init__(make, model, year) #herda atributos da classe-pai

#Instância
my_tesla = ElectricCar("tesla", "model s", "2016")
print(my_tesla.get_descriptive())
```

```
2016 Tesla Model S
```

O exemplo modela um carro elétrico. Um carro elétrico é apenas um tipo específico de carro, portanto podemos basear nossa nova classe `ElectricCar` na classe `Car` que escrevemos anteriormente. Então só precisaremos escrever o código para os atributos e os comportamentos específicos de carros elétricos.

Começamos com `Car`. Quando criamos uma classe-filha, a classe-pai deve fazer parte do arquivo atual e deve aparecer antes da classe-filha no arquivo. Definimos a classe-filha, que é `ElectricCar`. O nome da classe-pai deve ser incluído entre parênteses na definição da classe-filha. O método `__init__()` aceita as informações necessárias para criar uma instância de `Car`.
A função `super().` é uma função especial que ajuda o Python a criar conexões entre a classe-pai e a classe-filha. Essa linha diz ao Python para chamar o método `__init__()` da classe-pai de `ElectricCar`, que confere todos os atributos da classe-pai a `ElectricCar`. O nome super é derivado de uma conveção segundo a qual a classe-pai se chama *superclasse* e a classe-filha é a *subclasse*.
Ao criar uma instância da classe-filha, `my_tesla`, a linha chama o método `__init__()` definido em `ElectricCar` que, por sua vez, diz ao Python para chamar o método `__init__()` definido na classe-pai `Car`.

11.3.2 Herança em Python 2.7

Em Python 2.7, a herança pe um pouco diferente. A classe `ElectricCar` teria o seguinte aspecto:

```
#Herança

#classe-pai
class Car(object):
    """Uma tentativa simples de representar um carro."""

    #Inicializa atributos
    def __init__(self, make, model, year):
        """Inicializa os atributos que descrevem um carro."""
        self.make = make
        self.model = model
        self.year = year

        #Atributo default
        #Não precisa ser inicializado no parâmetro
        self.odometer_reading = 0 #Atributo com valor default

    #Métodos
    ##Descreve o carro atraves dos atributos
    def get_descriptive(self):
        """Devolve um nome descritivo, formatado de modo elegante."""
        long_name = str(self.year) + " "
        long_name += self.make + " "
        long_name += self.model
        return long_name.title()

    ##Imprime odometer na tela
    def read_odometer(self):
        """Exibe uma frase que mostra a milhagem do carro."""
        print("This car has " + str(self.odometer_reading) + " miles on it.")

    ##Método para modificar atributo
    def update_odometer(self,mileage):
        """
        Define o valor de leitura do hodômetro com o valor especificado.
        Rejeita aalteração se for tentativa de definir um valor menor para o hodômetro.
        """
        if mileage >= self.odometer_reading:
```

```
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    ##Método para incrementar atributo
    def increment_odometer(self, miles):
        """Soma a quantidade especificada ao valor de leitura do hodômetro."""
        if miles >= 0:
            self.odometer_reading += miles
        else:
            print("You can't roll back an odometer!")

#Classe-filha
class ElectricCar(Car):
    """Representa aspectos de um carro especificos de veiculos elétricos."""

    #Atributos herdados
    def __init__(self, make, model, year):
        """Inicializa os atributos da classe-pai."""
        super(ElectricCar,self).__init__(make, model, year)

#Instância
my_tesla = ElectricCar("tesla", "model s", "2016")
print(my_tesla.get_descriptive())
```

```
2016 Tesla Model S
```

A função `super()` precisa de dois argumentos: uma referência a classe-filha e o objeto `self`. Esses argumentos são necessários para ajudar o Python a fazer as conexões apropriadas entre as classes pai e filha. Quando usar herança em o Python 2.7, lembre-se de definir a classe-pai usando a sintaxe `object` também.

11.3.3 Definindo atributos e métodos da classe-filha

Depois que tiver uma classe-filha que herde de uma classe-pai, você pode adicionar qualquer atributo ou método novo necessários para diferenciar a classe-filha da classe-pai.
Exemplo:

```
#Herança

#classe-pai
class Car():
    """Uma tentativa simples de representar um carro."""

    #Inicializa atributos
    def __init__(self, make, model, year):
        """Inicializa os atributos que descrevem um carro."""
        self.make = make
        self.model = model
        self.year = year

        #Atributo default
        #Não precisa ser inicializado no parâmetro
        self.odometer_reading = 0 #Atributo com valor default

    #Métodos
    ##Descreve o carro atraves dos atributos
    def get_descriptive(self):
        """Devolve um nome descritivo, formatado de modo elegante."""
        long_name = str(self.year) + " "
        long_name += self.make + " "
        long_name += self.model
        return long_name.title()

    ##Imprime odometer na tela
    def read_odometer(self):
        """Exibe uma frase que mostra a milhagem do carro."""
        print("This car has " + str(self.odometer_reading) + " miles on it.")

    ##Método para modificar atributo
    def update_odometer(self,mileage):
        """
        Define o valor de leitura do hodômetro com o valor especificado.
        Rejeita aalteração se for tentativa de definir um valor menor para o hodômetro.
        """
```

```
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    ##Método para incrementar atributo
    def increment_odometer(self, miles):
        """Soma a quantidade especificada ao valor de leitura do hodômetro."""
        if miles >= 0:
            self.odometer_reading += miles
        else:
            print("You can't roll back an odometer!")

#Classe-filha
class ElectricCar(Car): #herança da classe-pai 'Car'
    """Representa aspectos de um carro especificos de veiculos elétricos."""

    #Atributos herdados
    def __init__(self, make, model, year):
        """
        Inicializa os atributos da classe-pai.
        Em seguida, inicializa os atributos especificos deum carro elétrico.
        """
        super().__init__(make, model, year) #herda atributos da classe-pai
        self.battery_size = 70 #definindo atrabuto da específico da classe-filha

    def describe_battery(self):
        """Exibe uma frase que descreve a capacidade da bateria."""
        print(f"This car has a {self.battery_size}-kWh battery.")

#Instância
my_tesla = ElectricCar("tesla", "model s", "2016")
print(my_tesla.get_descriptive())
my_tesla.describe_battery() #Chama atributo específico da classe-filha
```

```
2016 Tesla Model S
This car has a 70-kWh battery.
```

Adicionamos um novo atributo `self.battery_size` e definimos seu valor inicial, por exemplo, com 70. Esse atributo será associado a todas as instâncias criadas a partir da classe `ElectricCar`, mas não será associado a nenhuma instânciade `Car`. Também adicionamos um método chamado `describe_battery()`, que exibe informações sobre a bateria.

11.3.4 Sobrescrevendo métodos da classe-pai

Qualquer método da classe-pai que não se enquadre no que você estiver tentando modelar com a classe-filha pode ser sobrescrito. Para isso, defina um método na classe-filha com o mesmo nome do método da classe-pai que você deseja sobrescrever. O Python desprezará o método da classe-pai e só prestará atenção no método definido na classe-filha.

Suponha que a classe `Car` tenha um método chamado `fill_gas_tank()`. Esse método não faz sentido para um veículo totalmente elétrico, portanto você pode sobrescrever esse método. Exemplo:

```
#Herança

#classe-pai
class Car():
    """Uma tentativa simples de representar um carro."""

    #Inicializa atributos
    def __init__(self, make, model, year):
        """Inicializa os atributos que descrevem um carro."""
        self.make = make
        self.model = model
        self.year = year

        #Atributo default
        #Não precisa ser inicializado no parâmetro
        self.odometer_reading = 0 #Atributo com valor default
        self.tank_gas = False
    #Métodos
    ##Descreve o carro atraves dos atributos
    def get_descriptive(self):
        """Devolve um nome descritivo, formatado de modo elegante."""
        long_name = str(self.year) + " "
        long_name += self.make + " "
        long_name += self.model
        return long_name.title()

    ##Imprime odometer na tela
    def read_odometer(self):
        """Exibe uma frase que mostra a milhagem do carro."""
        print("This car has " + str(self.odometer_reading) + " miles on it.")

    ##Método para modificar atributo
```

```
    def update_odometer(self,mileage):
        """
        Define o valor de leitura do hodômetro com o valor especificado.
        Rejeita aalteração se for tentativa de definir um valor menor para o hodômetro.
        """
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    ##Método para incrementar atributo
    def increment_odometer(self, miles):
        """Soma a quantidade especificada ao valor de leitura do hodômetro."""
        if miles >= 0:
            self.odometer_reading += miles
        else:
            print("You can't roll back an odometer!")

    ##Método informa se tem um tank de gás
    def fill_gas_tank(self):
        if self.tank_gas == True:
            print("Contém um tanque de gás!")
        else:
            print("Não contém um tanque de gás!")

#Classe-filha
class ElectricCar(Car): #herança da classe-pai 'Car'
    """Representa aspectos de um carro especificos de veiculos elétricos."""

    #Atributos herdados
    def __init__(self, make, model, year):
        """
        Inicializa os atributos da classe-pai.
        Em seguida, inicializa os atributos especificos deum carro elétrico.
        """
        super().__init__(make, model, year) #herda atributos da classe-pai
        self.battery_size = 70 #definindo atrabuto da específico da classe-filha

    def describe_battery(self):
        """Exibe uma frase que descreve a capacidade da bateria."""
        print(f"This car has a {self.battery_size}-kWh battery.")
```

```
    #Sobrescrevendo método
    def fill_gas_tank(self):
        """Carros elétricos não tem tanques de gasolina."""
        print("This car doesn't need a gas tank!")

#Instância
my_tesla = ElectricCar("tesla", "model s", "2016")

#Chamando método sobrescrito
my_tesla.fill_gas_tank()
```

```
This car doesn't need a gas tank!
```

Se alguém tentar chamar `fill_gas_tank()` com um carro elétrico, o Python ignorará esse método de `Car` e executará o código da classe-filha, apresentado em seu lugar. Ao usar herança, você pode fazer suas classes-filhas preservarem o que for necessário e sobrescrever tudo o que não for utilizado da classe-pai.

11.3.5 Instâncias como atributos

Exemplo:

```
#Herança

#classes-pai
class Car():
    """Uma tentativa simples de representar um carro."""

    #Inicializa atributos
    def __init__(self, make, model, year):
        """Inicializa os atributos que descrevem um carro."""
        self.make = make
        self.model = model
        self.year = year

        #Atributo default
        #Não precisa ser inicializado no parâmetro
        self.odometer_reading = 0 #Atributo com valor default
        self.tank_gas = False
    #Métodos
    ##Descreve o carro atraves dos atributos
    def get_descriptive(self):
        """Devolve um nome descritivo, formatado de modo elegante."""
        long_name = str(self.year) + " "
        long_name += self.make + " "
        long_name += self.model
        return long_name.title()

    ##Imprime odometer na tela
    def read_odometer(self):
        """Exibe uma frase que mostra a milhagem do carro."""
        print("This car has " + str(self.odometer_reading) + " miles on it.")

    ##Método para modificar atributo
    def update_odometer(self,mileage):
        """
        Define o valor de leitura do hodômetro com o valor especificado.
        Rejeita aalteração se for tentativa de definir um valor menor para o hodômetro.
        """
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
```

```
        else:
            print("You can't roll back an odometer!")

    ##Método para incrementar atributo
    def increment_odometer(self, miles):
        """Soma a quantidade especificada ao valor de leitura do hodômetro."""
        if miles >= 0:
            self.odometer_reading += miles
        else:
            print("You can't roll back an odometer!")

    ##Método informa se tem um tank de gás
    def fill_gas_tank(self):
        if self.tank_gas == True:
            print("Contém um tanque de gás!")
        else:
            print("Não contém um tanque de gás!")

class Battery():
    """
    Uma tentativa de simples de modelar uma bateria de carro elétrico.
    """

    def __init__(self,battery_size=70):
        """Inicializa atributos da bateria."""
        self.battery_size = battery_size

    def describe_battery(self):
        """Exibe uma frase que descrevea capacidade da bateria."""
        print(f"This car has a {self.battery_size}-kWh battery.")

    def get_range(self):
        """
        Exibe uma frase sobre a distância
        que o carro é capaz de percorrer com essa bateria.
        """
        if self.battery_size == 70:
            range = 240
        elif self.battery_size == 85:
            range = 270

        message = "This car can go approximately " + str(range)
```

```
        message += " miles on a full charge."
        print(message)

#Classe-filha
class ElectricCar(Car): #herança da classe-pai 'Car'
    """Representa aspectos de um carro especificos de veiculos elétricos."""

    #Atributos herdados
    def __init__(self, make, model, year):
        """
        Inicializa os atributos da classe-pai.
        Em seguida, inicializa os atributos especificos deum carro elétrico.
        """
        super().__init__(make, model, year) #herda atributos da classe-pai
        self.battery = Battery() #Cria um atributo que é uma instância

#Instância
my_tesla = ElectricCar("tesla", "model s", "2016")

#Chamando método sobrescrito
print(my_tesla.get_descriptive())

#Chama atributo-instancia, que referencia um método
my_tesla.battery.describe_battery()
my_tesla.battery.get_range()
```

```
2016 Tesla Model S
This car has a 70-kWh battery.
This car can go approximately 240 miles on a full charge.
```

Definimos uma nova classe chamada `Battery` que não herda de nenhuma outra classe. O método `__init__()` tem um parâmetro, `battery_size`, além de `self`. É um parâmetro opcional que define a capacidade da bateria com 70 se nenhum valor for especificado. O método `describe_battery()` também foi transferido para essa classe.

Na classe `ElectricCar`, adicionamos um atributo chamado `self.battery`. Essa linha diz ao Python para criar uma nova instância de `Battery` (com capacidade default de 70, pois não estamos especificando nenhum valor) e armazenar essa instância no atributo `self.battery`. Isso acontecerá sempre que o método `__init__()` for chamado. Qualquer instância de `ElectricCar` agora terá uma instância de `Battery` criada automaticamente.

```
my_tesla.battery.describe_battery()
```

Essa linha diz ao Python para usar a instância `my_tesla`, encontra seu atributo `battery` e chama o método `describe_battery()` associado à instância de `Battery` armazenada no atributo.

Parece ser bastante trabalho extra, mas agora podemos descrever a bateria com quantos detalhes quisermos sem deixar a classe `ElectricCar` entulhada.

O novo método `get_range()` efetua uma análise simples. Se a capacidade da bateria for de 70kWh, `get_range()` define o alcance do carro com 240 milhas; se a capacidade for de 85kWh, o alcance será definido com 270 milhas. Esse valor é então apresentando. Quando quisermos usar esse método, novamente, devemos chamá-lo por meio do atributo `battery` do carro.

11.4 Importando classes

A medida que acrescentar mais funcionalidades as classes, seus arquivos ficarão maiores, mesmo quando usar herança de forma apropriada. Para estar de acordo com a filosofia geral do Python, quanto menos entulhados estiverem seus arquivos, melhor será. Para ajudar, o Python permite armazenar classes em módulos e então importar as classes necessárias em seu programa principal.

11.4.1 Importando uma única classe

O arquivo `car.py` somente com o código da classe `Car`:

```
"""Uma classe que pode ser usada para representar um carro."""

class Car():
    """Uma tentativa simples de representar um carro."""
    def __init__(self,make,model,year):
        """Inicializa os atributos que descrevem um carro."""
        #Atributo
        self.make = make
        self.model = model
        self.year = year

        #Atributo default
        self.odometer_reading = 0

    #Métodos
    def get_descriptive_name(self):
        """
        Devolve um nome descritivo formatado de modo elegante.
        """
        long_name = str(self.year) + " " + self.make
        long_name += " " + self.model
        return long_name.title()

    def read_odometer(self):
        """Exibe uma frase que mostra a milhagem do carro."""
        print(f"This car has {self.odometer_reading} miles on it.")

    def update_odometer(self,mileage):
        """
        Define o valor de leitura do hodômetro com o valor especificado.
```

```
        Rejeita alteração se for tentativa de definir um valor menor
        para o hodômetro.
        """
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self,miles):
        """
        Soma a quantidade especificada ao valor de leitura do hodômetro.
        """
        self.odometer_reading += miles
```

Incluímos uma `docstring` no nível de módulo que descreve rapidamente o conteúdo desse módulo. Escreva uma `docstring` para cada módulo que criar.

Agora criamos um arquivo separado chamado `my_car.py`. Esse arquivo importará a classe `Car` e então criará uma instância dessa classe:

```
from car import Car

#Chamadas
my_new_car = Car('audi', 'a4', 2016)
print(my_new_car.get_descriptive_name())

my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

A instrução `import` diz ao Python para abrir o módulo `car.py` e importar a classe `Car`. Agora podemos usar a classe `Car` como se ela estivesse definida nesse arquivo.

Importar classes é uma maneira eficiente de programar. Imagine o tamanho que esse programa teria se a classe `Car` inteira estivesse incluída. Quando transferimos a classe para um módulo e o importamos, continuamos usufruindo da mesma funcionalidade, porém o arquivo com o programa principal permanece limpo e fácil de ler. Também armazenamos a maior parte da lógica em arquivos separados; depois que suas classes estiverem funcionando conforme esperado, você poderá deixar de lado esses arquivos e se concentrar na lógica de mais alto nível de seu programa principal.

Sintaxe de importanção de uma única classe:
`from nome_módulo import nome_classe`
No nome do módulo, não colocamos para importar a extensão do arquivo (`.py`).

11.4.2 Armazenando várias classes em um módulo

Você pode armazenar tantas classes quantas forem necessárias em um único módulo, embora cada classe em um módulo deve estar, de algum modo, relacionada com outra classe. Ambas as classes, `battery` e `ElectricCar`, ajudam a representar carros, portanto vamos acrescentá-las ao módulo `Car.py`.

```
#Classes
class Car():
    """Uma tentativa simples de representar um carro."""
    def __init__(self,make,model,year):
        """Inicializa os atributos que descrevem um carro."""
        #Atributo
        self.make = make
        self.model = model
        self.year = year

        #Atributo default
        self.odometer_reading = 0

    #Métodos
    def get_descriptive_name(self):
        """
        Devolve um nome descritivo formatado de modo elegante.
        """
        long_name = str(self.year) + " " + self.make
        long_name += " " + self.model
        return long_name.title()

    def read_odometer(self):
        """Exibe uma frase que mostra a milhagem do carro."""
        print(f"This car has {self.odometer_reading} miles on it.")

    def update_odometer(self,mileage):
        """
        Define o valor de leitura do hodômetro com o valor especificado.
        Rejeita alteração se for tentativa de definir um valor menor
        para o hodômetro.
        """
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
```

```
            print("You can't roll back an odometer!")

    def increment_odometer(self,miles):
        """
        Soma a quantidade especificada ao valor de leitura do hodômetro.
        """
        self.odometer_reading += miles

class Battery():
    """Uma tentativa simples de modelar uma bateria para um carro elétrico."""

    #Atributos
    def __init__(self,battery_size=70):
        """Inicializa atributos da bateria."""
        self.battery_size = battery_size

    #Métodos
    def describe_battery(self):
        """Exibe uma fraseque descreve a capacidade da bateria."""
        print(f"This car has a {self.battery_size}-kWh battery.")

    def get_range(self):
        """
        Exibe frase sobre a distância que o carro pode percorrer
        com essa bateria.
        """
        if self.battery_size == 70:
            range = 240
        elif self.battery_size == 85:
            range = 270

        message = "This car can go approximately " + str(range)
        message += " miles on a full charge."
        print(message)

class ElectricCar(Car):

    #Atributos
    def __init__(self,make,model,year):
        """
        Inicializa os atributos da classe-pai.
        Em seguida, inicializa os atributos específicos de um carro elétrico.
        """
```

```
        super().__init__(make,model,year)
        self.battery = Battery()
```

Agora podemos criar um novo arquivo chamado `my_electric_car.py`, importar a classe `ElectricCar` e criar um carro elétrico.

```
#Importando classes
from car import ElectricCar

#Cria uma instância
my_tesla = ElectricCar("tesla",'model s',2016)

#Chama métodos
print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
my_tesla.battery.get_range()
```

Esse código gera a mesma saída que vimos antes, embora a maior parte da lógica esteja oculta em um módulo.

11.4.3 Importando várias classes de um módulo

Podemos importar quantas classes forem necessárias em um arquivo de programa.
Se quisermos criar um carro comum e um carro elétrico no mesmo arquivo, precisamos importar tanto a classe `Car` quanto a classe `ElectricCar`:

```
#Importando classes de um módulo
from car import Car, ElectricCar
#from nome_módulo import nome_classe_1, nome_classe_2, ...

#Instâncias
my_beetle = Car('volkswagem', 'beetle', 2016)
my_tesla = ElectricCar('tesla', 'roadster', 2016)

#Chamadas
print(my_beetle.get_descriptive_name())
print(my_tesla.get_descriptive_name())
```

Importe várias classes de um módulo separando cada classe com uma vírgula. Depois que importar as classes de que precisará, você poderá criar quantas instâncias de cada classe quantas forem necessárias.

11.4.4 Importando um módulo completo

Também podemos importar um módulo completo e então acessar as classes necessárias usando a notação de ponto. Essa abordagem é simples e resulta em um código fácil de ler. Como toda chamada que cria uma instância de uma classe inclui o nome do módulo, você não terá conflito de nomes com qualquer nome usado no arquivo atual.

Eis a aparência do código para importar o módulo `car` inteiro e então criar um carro comum e um carro elétrico:

```
#Importando um módulo completo
##Importa tudas as classes do módulo
import car
#from nome_módulo

#Instâncias
my_beetle = car.Car('volkswagem', 'beetle', 2016)
my_tesla = car.ElectricCar('tesla', 'roadster', 2016)

#Chamadas
print(my_beetle.get_descriptive_name())
print(my_tesla.get_descriptive_name())
```

Importamos o módulo `car` inteiro. Então acessamos as classes necessárias por meio da sintaxe: `nome_módulo.nome_da_classe`

11.4.5 Importando todas as classes de um módulo

Você pode importar todas as classes de um módulo usando a sintaxe:
`from nome_do_modulo import *`

Esse método não é recomendado por dois motivos. Em primeiro lugar, é conveniente ser capaz de ler as instruções `import` no início de um arquivo e ter uma noção clara de quais classes um programa utiliza. Com essa abordagem, não fica claro quais são as classes do módulo que você esta usando. Essa abordagem também pode resultar em confusão com nomes presentes no arquivo. Se você acidentalmente importar uma classe com o mesmo nome que outro item em seu arquivo de programa, poderá gerar erros dificeis de serem diagnoticados. Estou mostrando essa opção aqui porque, embora não seja uma abordagem recomendada, é provável que você vá vê-la no código de outras pessoas.

Se precisar importar muitas classes de um módulo, é melhor importar o módulo todo e usar a sintaxe `nome_do_modulo.nome_da_classe`. Você não verá todas as classes usadas no início do arquivo, mas verá claramente em que lugares o módulo é utilizado no programa. Possiveis conflitos de nomes que possam ocorrer ao importar todas as classes de um módulo também serão evitados.

11.4.6 Importando um módulo em um módulo

As vezes, você vai querer espalhar suas classes em vários módulos para impedir que um arquivo cresça demais e evitar a armazenagem de classes não relacionadas no mesmo módulo. Ao armazenar suas classes em vários módulos, você poderá descobrir que uma classe em um módulo depende de uma classe que está em outro módulo. Se isso acontecer, importe a classe necessária no primeiro módulo.

Por exemplo, vamos armazenar a classe `Car` em um módulo e as classes `ElectricCar` e `Battery` em um módulo separado. Criaremos um novo módulo chamado `electric_car.py` - substituindo o arquivo `electric_car.py` que criamos antes - e copiaremos apenas as classes `Battery` e `ElectricCar` para esse arquivo.

```
"""
Um conjunto de classes que pode ser usado para representar carros elétricos.
"""

#Importando classe-pai
from car import Car

#Classes-filhas
class Battery():
    """Uma tentativa simples de modelar uma bateria para um carro elétrico."""

    #Atributos
    def __init__(self,battery_size=70):
        """Inicializa atributos da bateria."""
        self.battery_size = battery_size

    #Métodos
    def describe_battery(self):
        """Exibe uma fraseque descreve a capacidade da bateria."""
        print(f"This car has a {self.battery_size}-kWh battery.")

    def get_range(self):
        """
        Exibe frase sobre a distância que o carro pode percorrer
        com essa bateria.
        """
        if self.battery_size == 70:
            range = 240
        elif self.battery_size == 85:
            range = 270
```

```
        message = "This car can go approximately " + str(range)
        message += " miles on a full charge."
        print(message)

class ElectricCar(Car):

    #Atributos
    def __init__(self,make,model,year):
        """
        Inicializa os atributos da classe-pai.
        Em seguida, inicializa os atributos específicos de um carro elétrico.
        """
        super().__init__(make,model,year)
        self.battery = Battery()
```

A classe `ElectricCar` precisa ter acesso a sua classe-pai `Car`, portanto importamos `Car` diretamente para o módulo. Se esquecermos de colocar essa linha, o Python gerará um erro quando tentarmos criar uma instância de `ElectricCar`. Também precisamos atualizar o módulo `Car` para que ele contenha apenas a classe `Car`.

```
"""
Uma classe que pode ser usada para representar um carro.
"""

#Classes
class Car():
    """Uma tentativa simples de representar um carro."""
    def __init__(self,make,model,year):
        """Inicializa os atributos que descrevem um carro."""
        #Atributo
        self.make = make
        self.model = model
        self.year = year

        #Atributo default
        self.odometer_reading = 0

    #Métodos
    def get_descriptive_name(self):
        """
        Devolve um nome descritivo formatado de modo elegante.
```

```
        """
        long_name = str(self.year) + " " + self.make
        long_name += " " + self.model
        return long_name.title()

    def read_odometer(self):
        """Exibe uma frase que mostra a milhagem do carro."""
        print(f"This car has {self.odometer_reading} miles on it.")

    def update_odometer(self,mileage):
        """
        Define o valor de leitura do hodômetro com o valor especificado.
        Rejeita alteração se for tentativa de definir um valor menor
        para o hodômetro.
        """
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self,miles):
        """
        Soma a quantidade especificada ao valor de leitura do hodômetro.
        """
        self.odometer_reading += miles
```

Agora podemos fazer a importação de cada módulo separadamente e criar o tipo de carro que for necessário.

```
#Importando classes dos módulos
from car import Car
from electric_car import ElectricCar, Battery

#Criando instâncias e chamadas de descrição da instância
my_beetle = Car("volkswagem", "beetle", 2016)
print(my_beetle.get_descriptive_name())

my_tesla = ElectricCar('tesla','roadster', 2016)
print(my_tesla.get_descriptive_name())
```

Importamos `Car` de seu módulo e `ElectricCar` de seu módulo. Em seguida, criamos um carro comum e um carro elétrico. Os dois tipos de carro são criados corretamente.

11.4.7 Definindo o seu próprio fluxo de trabalho

Quando estiver começando a programar, mantenha a estrutura de seu código simples. Procure fazer tudo em um só arquivo e transfira suas classes para módulos separados depois que tudo estiver funcionando. Se gostar do modo como os módulos e arquivos interagem, experimente armazenar suas classes em módulos quando iniciar um projeto. Encontre uma abordagem que permita escrever um código que funcione, e comece a partir daí.

11.5 Biblioteca-padrão do Python

A biblioteca-padrão de Python é um conjunto de módulos incluído em todas as instalações de Python. agora que temos uma compreensão básica de como as classes funcionam, podemos começar a usar módulos como esses, escritos por outros programadores. Podemos usar qualquer função ou classe da biblioteca-padrão incluindo uma instrução `import` simples no início do arquivo. Vamos analisar a classe `OrderedDict` do módulo `collections`.

Os dicionários permitem associar informações, mas eles não mantém um controle de ordem em que os pares `chave-valor` são acrescentados. Se você estiver criando um dicionário e quiser manter o controle da ordem em que os pares `chave-valor` são adicionados, a classe `OrderedDict` do módulo `collections` poderá ser usada. Instâncias da classe `OrderedDict` se comportam quase do mesmo modo que os dicionários, exceto que mantêm o controle da ordem em que os pares `chave-valor` são adicionados.

A classe `OrderedDict`, do módulo `collections`, é um dicionário que mantém a ordem dos pares `chave-valor`.

Vamos retornar o exemplo `favorite_languages.py`. Dessa vez, vamos manter o controle da ordem em que as pessoas responderam a enquete:

```
#Bibliotecas padrão do Python

#Importando biblioteca-padrão collections a função OrderedDict
#OrderedDict = é um dicionário que mantém a ordem dos pares chave-valor
from collections import OrderedDict

#Instância
favorite_languages = OrderedDict()

#Pares chave-valor
favorite_languages['jen']='python'
favorite_languages['sarah']='c'
favorite_languages['edward']='ruby'
favorite_languages['phil']='python'

#Mantem a ordem de entrada
for name, language in favorite_languages.items():
    print(f"{name.title()}'s favorite language is {language.title()}")
```

```
Jen's favorite language is Python
Sarah's favorite language is C
Edward's favorite language is Ruby
Phil's favorite language is Python
```

Começamos importando a classe `OrderedDict` do módulo `collections`. Criamos uma instância da classe `OrderedDict` e a armazenamos em `favorite_languages`. Observe que não há chaves; a chamada a `OrderedDict()` cria um dicionário ordenado vazio para nós e o armazena em `favorite_languages`. Então adicionamos cada nome e linguagem em `favorite_languages`, um de cada vez. Por fim, percorremos `favorite_languages` com um laço, sabemos que sempre teremos as respostas na ordem em que os itens foram adicionados.

É uma ótima classe para conhecer, pois combina a principal vantagem das listas (preservar a ordem original) com o principal recurso dos dicionários (associar informações). Quando começar a modelar situações do mundo real que sejam de seu interesse, é provável que você vá se deparar com uma situação em que um dicionário ordenado seja exatamente o que você precisa. A medida que conhecer melhor a biblioteca-padrão, você passará a ter familiariadade com vários módulos como esse, que ajudam a tratar situações comums.

Para conhecer outras bibliotecas-padrão do Python acesse:
https://pymotw.com/3/

11.6 Estilizando classes

Vale a pena esclarecer algumas questões ligadas a estilização de classes, em especial quando seus programas se tornarem mais complexos.

Os nomes das classes devem ser escritos com *CamelCaps*. Para isso, cada palavra do nome deve ter a primeira letra maiúscula, e você não deve usar underscores. Nomes de instâncias e de módulos devem ser escritos com letras minúsculas e underscores entre as palavras.

Toda classe deve ter uma docstring logo depois de sua definição. A docstring deve conter uma breve descrição do que a classe faz, e as mesmas convenções de formatação devem ser usadas para escrever docstrings em funções. Cada módulo também deve ter uma docstring que descreva para que servem as classes de um módulo.

Podemos usar linhas em branco para organizar o código, mas não deve utilizá-las de modo excessivo. Em uma classe, podemos usar uma linha em branco entre os métodos; em um módulo, podemos usar duas linhas em branco para separar as classes.

Se houver necessidade de importar um módulo da biblioteca-padrão e um módulo escrito por você, coloque a instrução de importação do módulo da biblioteca-padrão antes. Então acrescente uma linha em branco e a instrução de importação para o módulo que você escreveu. Em programas com várias instruções de importação, essa convenção facilita ver a origem dos diferentes módulos utilizados pelo programa.

12 Arquivos e exceções

Neste capítulo aprenderemos a trabalhar com arquivos para que seus programas possam analisar rapidamente muitos dados.

Veremos:

- Como tratar erros a fim de que seus programas não falhem quando se depararem com situações inesperadas. Conheceremos as *exceções* - objetos Python especiais para administrar erros que surgirem enquanto um programa estiver executando.
- Também conheceremos o módulo `json`, que permite salvar dados de usuários para que não sejam perdidos quando seu programa parar de executar.
- Aprenderemos a trabalhar com arquivos e a salvar dados deixará seus programas mais fáceis de usar. Os usuários poderão escolher quais dados devem fornecer e quando. As pessoas podem executar seu programa, fazer uma alguma tarefa e então fechá-lo e retomá-lo mais tarde, do ponto em que pararam.
- Aprender a tratar exceções ajudará você a lidar com situações em que os arquivos não existam e com outros problemas que possam fazer seus programas falharem. Isso deixará seus programas mais robustos quando dados ruins forem encontrados, sejam eles provenientes de erros inocentes ou de tentativas maliciosas de fazer seus programas falharem.

12.1 Lendo dados de um arquivo

Uma quantidade incrível de dados está disponível em arquivos de texto. Ler dados de um arquivo é particularmente útil em aplicações de análise de dados, mas também se aplica a quase qualquer situação em que você queira analisar ou modificar informações armazenadas em um arquivo. Por exemplo, podemos escrever um programa que leia o conteúdo de um arquivo-texto e reescreva o arquivo com uma formatação que permita a um navegador exibi-lo.

Quando quiser trabalhar com as informações de um arquivo-texto, primeiro passo será ler o arquivo em memória. Você pode ler todo o conteúdo de um arquivo ou pode trabalhar com uma linha de cada vez.

12.1.1 Lendo um arquivo inteiro

Para começar, precisamos de um arquivo com algumas linhas de texto. Vamos iniciar com um arquivo que contenha o valor de pi com trinta casas decimais, dez casas por linha:

```
3.14159265358979323846264338327
```

Salve o arquivo, em `.txt` no mesmo diretório em que você armazenará os programas deste capítulo.
Aqui esta um programa que abre esse arquivo, lê seus dados e exibe o conteúdo na tela:

```
#Abre arquivo, lê o conteudo e salva numa variável
#with fecha o arquivo depois de usa-lo
#open() abre o arquivo
#'as' salva o conteudo do arquivo no objeto file_object
with open('pi_digits.txt') as file_object:
    #Lê o conteúdo do arquivo, salva como string
    contents = file_object.read()

    #Elimina o caracter em branco do final
    contents = contents.rstrip()

    #Confere  tipo da variável, resultado esperado string
    print(type(contents))

    #Printa o conteúdo da variável = conteúdo do arquivo
    print(contents)
```

Muitas atividades acontecem na primeira linha desse programa. Vamos começar observando a função `open()`. Para realizar qualquer tarefa com um arquivo, mesmo que seja apenas exibir seu conteúdo, você precisará inicialmente abrir o arquivo para acessá-lo. A função `open()` precisa de um argumento: o nome do arquivo que você deseja abrir. Python procura esse arquivo no diretório em que o programa executando no momento está armazenado. Nesse exemplo, `file_reader.py` está executando, portanto o Python procura `pi_digits.txt` no diretório em que o arquivo `file_reader.py` esta armazenado. A função `open()` devolve um objeto que representa `digits.txt`. O Python armazena esse objeto em `file_object`, com o qual trabalharemos posteriormente no programa.

A palavra reservada `with` fecha o arquivo depois que não for mais necessário acessá-lo. Observe como chamamos `open()` nesse programa, mas não chamamos `close()`. Você pode abrir e fechar o arquivo chamando `open()` e `close()`, mas se um bug em seu programa impedir que a instrução `close()` seja executada, o arquivo não será fechado. Isso pode parecer trivial,

mas arquivos indevidamente fechados podem provocar perda de dados ou estes podem ser corrompidos. Além disso, se `close()` for chamado cedo de mais em seu programa, você se verá tentando trabalhar com um arquivo fechado (um arquivo que não pode ser acessado), o que resultará em erros. Nem sempre é fácil saber exatamente quando devemos fechar um arquivo, mas com a estrutura montrada aqui, o Python descobrirá isso para você. Tudo que você precisa fazer é abrir o arquivo e trabalhar com ele conforme desejado, com a confiança de que o Python o fechará automaticamente no momento certo.

Depois que tivermos um objeto arquivo que represente `pi_digits.txt`, usaremos o método `.read()` na segunda linha do nosso programa para ler todo o conteúdo do arquivo e armazená-lo em uma *longa string* em `contents`. Quando exibimos o valor de `contents`, vemos o arquivo-texto completo.
Exemplo do código:

```
contents = file_object.read()
```

O método `.read()` salva como tipo *string*, lembrar de verificar e/ou modificar o tipo antes de trabalhar com a variável.

A única diferença entre a saída e o arquivo original é a linha em branco extra no final da saída. A linha em branco aparece porque o método `.read()` devolve uma string vazia quando alcança o final do arquivo; essa string vazia aparece como uma linha em branco. Se quiser remover essa linha em branco extra, o método `.rstrip()` pode ser usada. Lembre-se de que o método `rstrip()` de Python remove qualquer caractere branco do lado direito de uma string. Agora a saída será exatamente igual ao conteúdo do arquivo original.
Exemplo do código:

```
contents = contents.rstrip()
```

12.1.2 Paths de arquivo

Quando um nome de arquivo simples como `pi_digits.txt` é passado para a função `open()`. O Python observa o diretório em que o arquivo executado no momento (isto é, seu arquivo de programa `.py`) esta armazenado.

As vezes, dependendo de como seu trabalho estiver organizado, o arquivo que você quer abrir não estará no mesmo diretório que o seu arquivo de programa. Por exemplo, você pode armazenar seus arquivos de programa em uma pasta chamada `python_work`; nessa pasta pode haver outra pasta chamada `text_files` para distinguir seus arquivos de programa dos arquivos-texto que eles manipulam. Apesar de `text_files` estar em `python_works`, simplesmente passar o nome de um arquivo que esta em `text_files` para `open()` não funcionará, pois o Python procurará o arquivo apenas em `python_works`; ele não prossegirá procurando em `text_files`. Para fazer o Python abrir arquivos de um diretório que não seja aquele em que seu arquivo de programa está armazenado, é preciso fornecer um *path de arquivo*, que diz ao Python para procurar em um local específico de seu sistema.

Como `text_files` está em `python_works`, você pode usar um *path de arquivo relativo* para abrir um arquivo em `text_files`. Um path de arquivo relativo diz ao Python para procurar um local especificado, relativo ao diretório em que o arquivo de programa em execução no momento esta armazenado.

No Linux e no OS X, você escreveria da seguinte forma:

```
with open('text_files/nome_do_arquivo.txt') as file_object:
```

Essa linha diz ao Python para procurar o arquivo `.txt` desejado na pasta `text_files` e supõem que essa pasta esta localizada em `python_works` (e está).

Em sistemas Windows, use uma barra invertida () no lugar da barra para a frente (/) no path do arquivo:

```
with open('text_files\nome_do_arquivo.txt') as file_object:
```

Você também pode dizer ao Python exatamente em que local está o arquivo em seu computador, não importa o lugar em que o programa em execução no momento esteja armazenado. Isso é chamado de *path absoluto* do arquivo. Utilize um path absoluto se um path relativo não funcionar. Por exemplo, se você colocou `text_files` em alguma pasta diferente de `python_works` - por exemplo, em uma pasta chamada `other_files` - então simplesmente passar o path 'text_files/nome_do_arquivo.txt' para `open()` não funcionará porque o Python procurará esse local somente em `python_works`. Você precisará fornecer um path completo para deixar claro em que lugar você quer que o Python procure.

Paths absolutos geralmente são mais longos que paths relativos, portanto é conveniente armazená-los em uma variável e então passar essa variável para `open()`.

No Linux e no OS X, paths absolutos tem o seguinte aspectos:

```
file_path = '/home/ehmatthes/other_files/text_files/nome_do_arquivo.txt'
with open(file_path) as file_object:
```

No Windows, eles se parecem com:

```
file_path = 'C:\Users\ehmatthes\other_files\text_files\nome_do_arquivo.txt'
with open(file_path) as file_object:
```

Ao usar paths absolutos, podemos ler arquivos em qualquer lugar do sistema. Por enquanto, é mais fácil armazenar arquivos no mesmo diretório em que estão seus arquivos de programas ou em uma pasta como `text_files` no diretório em que estiverem seus arquivos de programa.

Sistemas Windows as vezes interpretam corretamente as barras para a frente nos paths de arquivo. Se você usa Windows e não esta obtendo os resultados esperados, tente usar barras invertidas.

Tipos de path de arquivo:

1. *path de arquivo relativo*
   Diz ao Python para procurar o arquivo desejado na pasta especificada e supõem que essa pasta esta localizada dentro do diretório do programa em execução. Usa o diretório do programa em execução como referência.
2. *path absoluto do arquivo*
   Diz ao Python exatamente em que local está o arquivo em seu computador, não importa o lugar em que o programa em execução no momento esteja armazenado. Fornecer um path completo para deixar claro em que lugar você quer que o Python procure.

12.1.3 Lendo dados linha a linha

Método `for` para ler linha por linha de um arquivo.

Quando estiver lendo um arquivo, com frequência você vai querer analisar cada linha do arquivo. Talvez você esteja procurando determinada informação no arquivo ou queira modificar o texto do arquivo de alguma maneira. Por exemplo, que inclua a palavra ensolarado ma descrição da previsão do tempo para esse dia. Em uma notícia, talvez queira procurar todas as linhas com a tag `<headline>` e reescrever essa linha com um tipo específico de formação.

Podemos usar um laço `for` no objeto arquivo para analisar cada uma de suas linhas, uma de cada vez:

```
#Path relativo
file_name = 'text_files/pi_digits.txt'

#Abrir arquivo pi_digits.txt
with open(file_name) as file_object:

    #Método for para ler linhas de um arquivo
    for line in file_object:
        print(line.rstrip())
```

Armazenamos o nome do arquivo que estamos lendo em uma variável `filename` (caminho ate o arquivo). Essa é uma convenção comum quando trabalhamos com arquivos. Como a variável `filename` não representa o arquivo propriamente dito - é apenas uma string que diz ao Python em que lugar o arquivo se encontra - você pode facilmente trocar pelo nome de outro arquivo com o qual você queira trabalhar. Depois da chamada a `open()`, um objeto que representa o arquivo e seu conteúdo e armazenado na variável `file_object`. Novamente, usamos a sintaxe `with` para deixar o Python abrir e fechar o arquivo de modo apropriado. Para analisar o conteúdo do arquivo, trabalhamos com cada linha do arquivo percorrendo o objeto arquivo em um laço.

Quando exibimos cada linha, encontramos outros linhas em branco. Essas linhas em branco aparecem porque um caractere invisível de quebra de linha esta no final de cada linha do arquivo-texto. A instrução `print` adiciona a sua propria quebra de linha sempre que a chamamos, portanto acabamos com dois caracteres de quebra de linha no final de cada linha: um do arquivo e outro da instrução `print`. Se usarmos `rstrip()` em cada linha na instrução `print`, eliminamos essas linhas em branco extras.

```
print(line.rstrip())
```

Agora a saída será igual ao conteúdo do arquivo novamente.

12.1.4 Criando uma lista de linhas de um arquivo - `.readlines()`

O método `.readlines()` armazena cada linha do arquivo em uma lista.

Quando usamos `with`, o objeto arquivo devolvido por `open()` estará disponível somente no bloco `with` que o contém. Se quiser preservar o acesso ao conteúdo de um arquivo fora do bloco `with`, você pode armazenar as linhas do arquivo em uma lista dentro do bloco e então trabalhar em essa lista. Pode processar partes do arquivo imediatamente e postergar parte do processamento de modo que seja feita mais tarde no programa.

O exemplo a seguir armazena as linhas de `pi_digits.txt` em uma lista no bloco `with` e, em seguida, exibe as linhas fora desse bloco:

```
#Path relativo
file_name = 'text_files/pi_digits.txt'

#Abrir arquivo pi_digits.txt
with open(file_name) as file_object:

    #Método readlines, para leitura de linhas em uma lista
    lines = file_object.readlines()

#Percorrendo a lista de linhas
for line in lines:
    print(line.rstrip()) #Elimando espaços em branco com o método rstrip()
```

O método `.readlines()` armazena cada linha do arquivo em uma lista. Essa lista é então armazenada em `lines`, com a qual podemos continuar trabalhando depois que o bloco `with` terminar. Usamos um laço `for` simples para exibir cada linha de `lines`. Como cada item de `lines` corresponde a uma linha do arquivo, a saída será exatamente igual ao conteúdo do arquivo.

12.1.5 Trabalhando com o conteúdo de um arquivo

Depois de ler o arquivo em memória, você pode fazer o que quiser com esses dados. Desse modo, vamos explorar rapidamente os dígitos de pi. Em primeiro lugar, tentaremos criar uma única string contendotodos os dígitos do arquivo, sem espaços em branco.
Exemplo:

```
#path, caminho ate o arquivo
file_name = 'text_files/pi_digits.txt'

with open(file_name) as file_object: #Abre arquivo
    lines = file_object.readlines() #Cria um lista de linhas

pi_string = '' #Inicializa string pi como vazia
for line in lines: #Pecorre as linhas
    pi_string += line.rstrip() #Junta strings e remove espaços em branco

print(pi_string) #string pi sem espaço em branco
print(len(pi_string)) #Tamanho da string
```

Começamos abrindo o arquivo e armazenando cada linha de dígitos em uma lista, como fizemos no exemplo anterior. Criamos uma variável `pi_string` para armazenar os dígitos de pi. Então criamos um laço que acrescenta cada uma das linhas de dígitos em `pi_string` removendo o caractere de quebra de linha. Exibimos essa string e mostramos também o seu tamanho.

Podemos nos livrar dos espaços em branco usando `strip()` ou `rstrip()`.

Quando Python lê um arquivo-texto, todo o texto do arquivo é interpretado como uma string. Se você ler um número e quiser trabalhar com esse valor em um contexto numérico, será necessário convertê-lo em um número de ponto flutuante com a função `float()`.

12.1.6 Arquivos grandes: um milhão de dígitos

Até agora, nosso enfoque foi analisar um arquivo-texto que contenha apenas três linhas, mas o código desses exemplos também funcionará bem em arquivos muito maiores. Se começarmos com um arquivo-texto que contenha pi com um milhão de casas decimais, e não trinta, uma única string contendo todos esses dígitos poderá ser criada. Não precisamos alterar nada em nosso programa, exceto para lhe passar um arquivo diferente. Além disso, exibiremos as cinquenta primeiras casas decimais para que não seja necessário assistirmos a um milhão de dígitos rolando pelo terminal.

```
#path, caminho ate o arquivo
file_name = 'text_files/pi_million_digits.txt'

with open(file_name) as file_object: #Abre arquivo
    lines = file_object.readlines() #Cria um lista de linhas

pi_string = '' #Inicializa string pi como vazia
for line in lines: #Pecorre as linhas
    pi_string += line.strip() #Junta strings e remove espaços em branco

print(pi_string[:50] + "...") #string pi sem espaço em branco
print(len(pi_string)) #Tamanho da string
```

O Python não tem nenhum limite inerente para a quantidade de dados com que podemos trabalhar; podemos trabalhar com tantos dados quantos a memória de seu sistema for capaz de tratar.

Arquivos para baixar: https://ehmatthes.github.io/pcc_2e/regular_index/

12.1.7 Seu aniversário esta contido em pi?

Vamos usar o programa que acabamos de escrever para descobrir se a data de nascimento de alguém aparece em algum ponto no primeiro milhão de dígitos de pi. Podemos fazer isso expressando cada data de nascimento como uma string de dígitos e verificando se essa string aparece em algum ponto de pi_string.

```
#path, caminho ate o arquivo
file_name = 'text_files/pi_million_digits.txt'

with open(file_name) as file_object: #Abre arquivo
    lines = file_object.readlines() #Cria um lista de linhas

pi_string = '' #Inicializa string pi como vazia
for line in lines: #Pecorre as linhas
    pi_string += line.strip() #Junta strings e remove espaços em branco

birthday = input("Enter your birthday, in the form mmddyy: ")
if birthday in pi_string:
    print("Your birthday appears in the first million digits of pi!")
else:
    print("Your birthday does not appear in the first million digits of pi!")
```

12.2 Escrevendo dados em um arquivo

Uma das maneiras mais simples de salvar dados é escrevê-los em um arquivo. Quando um texto é escrito em um arquivo, o resultado estará disponível depois que você fechar o terminal que contém a saída de seu programa. Podemos analisar a saída depois que um programa acabar de executar e compartilhar os arquivos de saída com outras pessoas também. Além disso, podemos escrever programas que leiam o texto de volta para a memória e trabalhar com esses dados novamente.

12.2.1 Escrevendo dados em um arquivo vazio

Para escrever um texto em um arquivo, chame `open()` com um segundo argumento que diga ao Python que você quer escrever dados no arquivo.
Exemplo:

```
#Nome do arquivo a ser criado, se não existir
filename = 'programming.txt'

#Abre e define que é um arquivo a ser criado e escrito ('w')
with open(filename,'w') as file_object:
    file_object.write("I love programming.") #Escreve no arquivo
```

A chamada `open()` nesse exemplo tem dois argumentos. O primeiro argumento ainda é o nome do arquivo que queremos abrir. O segundo argumento 'w', diz ao Python que queremos abrir o arquivo no *modo de escrita*. Podemos abrir um arquivo em *modo de leitura* ('r'), em *modo de escrita* ('w'), em *modo de concatenação* ('a') ou em um modo que permita ler e escrever no arquivo ('r+').

Se o argumento por omitido, por padrão o Python abrirá o arquivo em modo somente de leitura.

A função `open()` cria automaticamente o arquivo no qual você vai escrever caso ele ainda não exista. No entanto, tome cuidado ao abrir um arquivo em modo de escrita ('w') porque se o arquivo já existir, o Python apagará antes de devolver o objeto arquivo (apaga o arquivo existente e cria outro).

Usamos o metódo `write()` no objeto arquivo para escrever uma string nesse arquivo. Esse programa não tem saída no terminal, mas se abrir o arquivo `programming.txt`, você verá uma linha de texto.

Esse arquivo se comporta como qualquer outro arquivo de seu computador. Você pode abri-lo, escrever um novo texto nele, copiar ou colar dados e assim por diante.

O Python escreve apenas strings em um arquivo-texto. Se quiser armazenar dados númericos em um arquivo-texto, será necessário converter os dados em um formato de string antes usando a função `str()` (converte para string).

Modos de abrir um arquivo:

- Modo de leitura (‘r’)
- Mode de escrita (‘w’)
- Modo de concatenação (‘a’)
- Modo de leitura e escrita (‘r+’)

12.2.2 Escrevendo várias linhas

A função `write()` não acrescenta nenhuma quebra de linha ao texto que você escrever. Portanto, se escrever mais de uma linha sem incluir caracteres de quebra de linha, seu arquivo poderá não ter a aparencia desejada. Ao abrir o arquivo `programming.txt`, você verá duas linhas emendadas.

A inclusão de quebra de linha em suas instruções `write()` faz cada string aparecer em sua propria linha.
Exemplo:

```
#Nome do arquivo a ser criado
filename = 'programming.txt'

#Abre e define que é um arquivo a ser criado e escrito ('w')
with open(filename,'w') as file_object:
    file_object.write("I love programming.\n") #Insere quebra de linha
    file_object.write("I love creating new games.\n")
```

A saída agora aparece em linhas separadas.

Podemos também usar espaços, caracteres de tabulação e linhas em branco para formatar a saída, exatamente como viemos fazendo com as saídas no terminal.

12.2.3 Concatenando dados em um arquivo

Se quiser acrescentar conteúdos em vez de sobrescrever o conteúdo existente, você pode abrir o arquivo em *modo de concatenação*. Ao abrir um arquivo em modo de concatenação, o Python não apagará o arquivo antes de volver o objeto arquivo. Qualquer linha que você escrever no arquivo será adicionada no final. Se o arquivo ainda não existe, o Python criará um arquivo vazio para você.

Vamos modificar `write_message.py`:

```
#Nome do arquivo
filename = 'programming.txt'

#Abre o arquivo e concatena dados ('a'), acrescenta informação
with open(filename,'a') as file_object:
    file_object.write("I also finding meaning in large datasets.\n")
    file_object.write("I love creating apps that can run in a browser.\n")
```

Usamos o argumento 'a' para abrir o arquivo para concatenação, em vez de sobrescrever o arquivo existente. Escrevemos duas linhas novas, que são acrescentadas em `programming.txt`.

Ao final, temos o conteúdo original do arquivo, seguido de um novo conteúdo que acabamos de acrescentar.

12.3 Exceções

O Python usa objetos especiais chamados `exceções` para administrar erros que surgirem durante a execução de um programa. Sempre que ocorrer um erro que faça o Python não ter certeza do que deve fazer em seguida, um objeto exceção será criado. Se você escrever um código que trate a exceção, o programa continuará executando. Se a exceção não for tratada, o programa será interrompido e um *traceback*, que inclui uma informação sobre a exceção levantada, será exibida.

As exceçõessão tratadas como blocos `try-except`. Um bloco `try-except` pede que o Python faça algo, mas também lhe diz o que deve ser feito se uma exceção for levantada. Ao usar blocos `try-except`, seus programas continuarão a executar, mesmo que algo comece a dar errado. Em vez de *tracebacks*, que podem ser confusos para os usuários lerem, os usuários verão mensagens de erro simpáticas escritas por você.

12.3.1 Tratando a exceção `ZeroDivisionError`

Vamos observar um erro simples, que faz o Python levantar uma exceção. Provavelmente, você sabe que é impossível dividir um número por zero, mas vamos pedir que o Python faça isso, de qualquer modo.
Exemplo:

```
print(5/0)
```

É claro que o Python não pode fazer essa operação, portanto veremos um traceback.

```
Traceback (most recent call last):

  File ~/anaconda3/lib/python3.11/site-packages/spyder_kernels/customize/utils.py:209 in exec
    exec_fun(compile(code_ast, filename, "exec"), globals)

  File ~/Programacao/python/Cap10-Arquivos_e_excecoes/03-Excecoes/division.py:9
    print(5/0)

ZeroDivisionError: division by zero
```

O erro informado no traceback, `ZeroDivisionError`, é um objeto exceção. O Python cria esse tipo de objeto em resposta a uma situação em que ele não é capaz de fazer o que lhe pedimos. Quando isso acontece, o Python interrompe o programa e informa o tipo de exceção levantada. Podemos usar essa informação para modificar nosso programa.

Diremos ao Python o que ele deve fazer quando esse tipo de exceção ocorrer; desse modo, se ela ocorrer novamente, estaremos preparados.

12.3.2 Usando blocos `try-except`

Quando achar que um erro pode ocorrer, você poderá usar um bloco `try-except` para tratar a exceção possível de ser levantada. Dizemos ao Python para tentar executar um código e lhe dizemos o que fazer caso o código resulte em um tipo particular de exceção.

```
try:
    print(5/0)
except ZeroDivisionError:
    print("You can´t divide by zero!")
```

```
You can´t divide by zero!
```

Colocamos `print(5/0)` - a linha que causou o erro - em um bloco `try`. Se o código em um bloco `try` funcionar, o Python ignorará o bloco `except`. Se o código no bloco `try` causar um erro, o interpretador procurará um bloco `except` cujo erro coincida com aquele levantado e executará o código desse bloco.

Nesse exemplo, o código no bloco `try` gera um `ZeroDivisionError`, portanto o Python procura um bloco `except` que lhe diga como deve responder. O interpretador então executa o código desse bloco e o usuário vê uma mensagem de erro simpática no lugar de um traceback.

Se houver mais código depois do bloco `try-except`, o programa continuará executando, pois dissemos ao Python como o erro deve ser tratado.

12.3.3 Usando exceções para evitar falhas

Tratar erros de forma correta é importante, em especial quando o programa tiver outras atividades para fazer depois que o erro ocorrer. Isso acontece com frequência em programas que pedem dados de entrada aos usuários. Se o programa responder a entradas inválidas de modo apropriado, ele poderá pedia mais entradas válidas em vez de causar uma falha.

Vamos criar uma calculadora simples que faça apenas divisões.
Exemplo:

```
print("Give me two numbers, and I'll divide them.")
print("Enter 'q' to quit.")

while True:
    first_number = input("\nFirst number: ")
    if first_number == 'q':
        break
    second_number = input("\nSecond number: ")
    if second_number == 'q':
        break
    answer = int(first_number)/int(second_number)
    print(answer)
```

Esse programa pedeque o usuário forneça o primeiro número (`first_number`) e, se o usuário não digitar 'q' para sair, pede um segundo número (`second_number`). Então dividimos esses dois números para obter a resposta (`answer`). O programa não faz nada para tratar erros, portanto pedir que uma divisão por zero seja feita causará uma falha do programa.

O fato do programa falhar é ruim, mas também não é uma boa ideia deixar que os usuários vejam o *tracebacks*. Usuários que não sejam técnicos ficarão confusos com eles e, em um ambiente malicioso, invasores aprenderão mais do que você quer que eles saibam a partir de um *traceback*. Por exemplo, eles saberão o nome de seu arquivo de programa e verão uma parte do código que não está funcionando de forma apropriada. As vezes, um invasor habilidoso pode usar essas informações para determinar os tipos de ataques que podem usar contra o seu código.

12.3.4 Bloco `else`

Podemos deixar esse programa mais resistente a erros colocando a linha capaz de produzir erros em um bloco `try-except`. O erro ocorre na linha que calcula a divisão, portanto é ai que colocaremos o bloco `try-except`. Esse exemplo também inclui um bloco `else`. Qualquer código que dependa do bloco `try` executar com sucesso deve ser colocado no bloco `else`.
Exemplo:

```
print("Give me two numbers, and I'll divide them.")
print("Enter 'q' to quit.")

while True:
    first_number = input("\nFirst number: ")
    if first_number == 'q':
        break
    second_number = input("\nSecond number: ")
    if second_number == 'q':
        break
    try: #Tentar fazer
        answer = int(first_number)/int(second_number)
    except ZeroDivisionError: #Error
        print("You can't divide by 0!")
    else: #Sucesso
        print(answer)
```

- Caso de sucesso, bloco `else`:

  Pedidos ao Python para tentar concluir a operação de divisão em um bloco `try`, que inclui apenas o código que pode causar um erro. Qualquer código que dependa do sucesso do bloco `try` é adicionado no bloco `else`. Nesse caso, se a operação de divisão for bem-sucedida, usamos o bloco `else` para exibir o resultado.

- Caso de erro, bloco `except`:

  O bloco `except` diz como o Python deve responder quando um `ZeroDivisionError` ocorrer. Se a instrução `try` não for bem-sucedida por causa de um erro de divisão por zero, mostraremos uma mensagem simpática informando o usuário de que modo esse tipo de erro pode ser evitado. O programa continua executando e o usuário jamais verá um *traceback*.

- Bloco `try-except-else`:

  O bloco `try-except-else` funciona assim: o Python tenta executar o código que esta na instrução `try`. O único código que deve estar em uma instrução `try` é aquele que pode fazer uma exceção ser levantada. As vezes, você terá um código adicional que deverá ser

executado somente se o bloco `try` tiver sucesso; esse código deve estar no bloco `else`. O bloco `except` diz ao Python o que ele deve fazer, caso uma determinada exceção ocorra quando ele tentar executar o código que esta na instrução `try`.

- O ganho usando bloco `try-except-else`:

  Ao prever possíveis fontes de erros, podemos escrever programas robustos, que continuarão a executar mesmo quando encontrarem dados inválidos ou se depararem com recursos ausentes. Seu código será resistente a erros inocentes do usuário e a ataques maliciosos.

- **Resumo** bloco `try-except-else`:

  - `try`

    Parte do código que pode fazer uma exceção ser levantada.

  - `except`

    Diz ao Python o que deve fazer, caso uma determinda exceção ocorra quando ele tentar executar o código na instrução `try`.

  - `else`

    Um código adicional que deverá ser executado somente se o bloco `try` tiver sucesso. Esse código deve estar no bloco `else`.

12.3.5 Tratando a exceção `FileNotFoundError`

Um problema comum ao trabalhar com arquivos é o tratamento de arquivos ausentes. O arquivo que você está procurando pode estar em outro lugar, o nome do arquivo pode estar escrito de forma incorreta ou o arquivo talvez simplesmente não exista. Podemos tratar todas essas situações de um modo simples com um bloco `try-except`.

Vamos tentar ler um arquivo que não existe. O programa a seguir tenta ler o conteúdo de *Alice in wonderland*, mas não salvei o arquivo `alice.txt` no mesmo diretório em que esta `alice.py`.

```
filename = 'alice.txt'

with open(filename) as f_obj:
    contents = f_obj.read()
```

O Python não é capaz de ler um arquivo ausente, portanto uma exceção é levantada:

```
Traceback (most recent call last):

  File ~/anaconda3/lib/python3.11/site-packages/spyder_kernels/customize/utils.py:209 in exec
    exec_fun(compile(code_ast, filename, "exec"), globals)

  File ~/Programacao/python/Cap10-Arquivos_e_excecoes/03-Excecoes/alice.py:11
    with open(filename) as f_obj:

FileNotFoundError: [Errno 2] No such file or directory: 'alice.txt'
```

A última linha do *traceback* informa um `FileNotFoundError`: essa é a exceção criada pelo Python quando não encontra o arquivo que está tentando abrir. Nesse exemplo, a função `open()` gera o erro, portanto, para tratá-lo, o bloco `try` tem início imediatemente antes da linha que contém essa função.

```
filename = 'alice.txt'

try:
    with open(filename) as f_obj:
        contents = f_obj.read()
except FileNotFoundError:
    msg = "Sorry, the file " + filename + " does not exist."
    print(msg)
```

Nesse exemplo, o código no bloco `try` gera um `FileNotFoundError`, portanto o Python procura um bloco `except` que trate esse erro. O interpretador então executa o código que está nesse bloco e o resultado é uma mensagem de erro simpática no lugar de um *traceback*.

12.3.6 Analisando textos - método `.split()`

Os textos usados nesta seção foram extraídos do projeto Gutenberg (https://gutenberg.org/).

Vamos obter o texto de *Alice in Wonderland* e tentar contar o número de palavras do texto. Usaremos o método string `split()`, que cria uma lista de palavras a partir de uma string. Eis o que `split()` faz com uma string que contém apenas o título "Alice in Wonderland".

```
#Conta o número de palavras do título do livro
title = "Alice in Wonderland"

words = title.split()

print(words)
```

```
['Alice', 'in', 'Wonderland']
```

O método `split()` separa uma string em partes sempre que encontra um espaço, e armazena todas as partes da string em uma lista. O resultado é uma lista de palavras da string, embora algumas pontuações possam também aparecer com determinadas palavras. Para contar o número de palavras em "Alice in Wonderland", usaremos `split()` no texto todo. Em seguida, contaremos os itens da lista para ter uma ideia geral da quantidade de palavras no texto.

```
filename = 'alice.txt'

try:
    with open(filename) as f_obj:
        contents = f_obj.read()
except FileNotFoundError:
    msg = "Sorry, the file " + filename + " does not exist."
    print(msg)
else:
    #Conta o número aproximado de palavras no arquivo
    words = contents.split() # Cria uma lista com as palavras do texto
    num_words = len(words) # Conta o número de palavras
    print(f"The file {filename} has about {num_words} words.")
```

Movi o arquivo alice.txt para o diretório correto, portanto o bloco `try` funcionará dessa vez. Usamos a string `contents`, que agora contém todo o texto de *Alice in Wonderland* como uma string longa, e aplicamos o método `split()` para obter uma lista de todas as palavras do livro.

Quando usamos `len()` nessa lista para verificar o seu tamanho, obtemos uma boa aproximação do número de palavras na string original. Exibimos uma frase que informa quantas palavras encontramos no arquivo. Esse código é colocado no bloco `else` porque funcionará somente se o código no bloco `try` for executado com sucesso. A saída nos informa quantas palavras estão em `alice.txt`.

12.3.7 Trabalhando com vários arquivos

Vamos acrescentar outros livros para analisar. Porém, antes disso, vamos passar a parte principal desse programa para uma função chamada `count_words()`. Com isso, será mais fácil a análise para diversos livros.

```
def count_words(filename):
    """Conta o número aproximado de palavras em um arquivo."""
    try:
        with open(filename) as f_obj:
            contents = f_obj.read()
    except FileNotFoundError:
        msg = "Sorry, the file " + filename + " does not exist."
        print(msg)
    else:
        #Conta o número aproximado de palavras no arquivo
        words = contents.split() # Cria uma lista com as palavras do texto
        num_words = len(words) # Conta o número de palavras
        print(f"The file {filename} has about {num_words} words.")

filenames = ['alice.txt','siddhartha.txt','moby_dick.txt','little_women.txt']
for filename in filenames:
    count_words(filename)
```

A maior parte desse código não foi alterada. Simplesmente, indentamos o código e o movemos para o corpo de `count_words()`. Manter os comentários atualizados é um bom hábito quando modificamos um programa; assim, transformamos o comentário em uma docstring e o alteramos um pouco.

Agora podemos escrever um laço simples para contar as palavras de qualquer texto que quisermos analisar. Fazemos isso armazenando os nomes dos arquivos que desejamos analisar em uma lista e, em seguida, chamando `count_words()` para cada arquivo da lista. Vamos experimentar contar as palavras das obras *Alice in Wonderland*, *Siddhartha*, *Moby Dick* e *Little Women*, todas disponiveis em dominio público. Deixei `siddhartha.txt` fora do diretório que contém `word_count.py` de propósito para que possamos ver como nosso programa trata um arquivo ausente de modo apropriado.

O arquivo `siddhartha.txt` ausente não tem efeito algum no restante da execução do programa.

O uso do bloco `try-except` nesse exemplo oferece duas vantagens significativas. Evitar que nossos usuários vejam um traceback e deixamos o programa continuar a análise dos textos que puder encontrar. Se não capturássemos e erro `FileNotFoundError` gerado por `siddhartha.txt`, o usuário veria um traceback completo e o programa pararia de executar após tentar analisar *Siddhartha*. *Moby Dick* e *Little Women* não seriam analisados.

12.3.8 Falhando silenciosamente - instrução `pass`

No exemplo anterior, informamos nossos usuários que um dos arquivos estava indisponível. Porém, não precisamos informar todas as exceções capturadas. As vezes, queremos que o programa falhe silenciosamente quando uma exceção ocorrer e continue como se nada tivesse acontecido. Para fazer um programa falhar em silêncio, escreva um bloco `try` como seria feito normalmente, mas diga de forma explícita ao Python para não fazer nada no bloco `except`. O Python tem uma instrução `pass` que lhe diz para não fazer nada em um bloco.

```
def count_words(filename):
    """Conta o número aproximado de palavras em um arquivo."""
    try:
        with open(filename) as f_obj:
            contents = f_obj.read()
    except FileNotFoundError:
        pass #Falhando silenciosamente
    else:
        #Conta o número aproximado de palavras no arquivo
        words = contents.split() # Cria uma lista com as palavras do texto
        num_words = len(words) # Conta o número de palavras
        print(f"The file {filename} has about {num_words} words.")

filenames = ['alice.txt','siddhartha.txt','moby_dick.txt','little_women.txt']
for filename in filenames:
    count_words(filename)
```

A única diferença entre essa listagem e a listagem anterior está na instrução `pass`. Agora, quando um `FileNotFoundError` é levantado, o código no bloco `except` é executado, mas nada acontece. Nenhum traceback é gerado e não há nenhuma saída em resposta ao erro levantado. Os usuários veem os contadores de palavras para cada arquivo existente, mas não há indicação sobre um arquivo não encontrado.

A instrução `pass` também atua como um marcador. É um lembrete de que você optou por não fazer nada em um ponto específico da execução de seu programa, mas talvez queira fazer algo nesse local, no futuro. Por exemplo, nesse programa, podemos decidir escrever os nomes de qualquer arquivo ausente em um arquivo chamado `missing_files.txt`. Nossos usuários não verão esse arquivo, mas poderemos lê-lo e tratar qualquer texto ausente.

12.3.9 Decidindo quais erros devem ser informados

Como sabemos quando devemos informar um erro aos usuários e quando devemos falhar silenciosamente? Se os usuários souberem quais textos devem ser analisados, poderão apreciar uma mensagem que lhes informe por que alguns textos não foram analisado. Se os usuários esperam ver alguns resultados, mas não sabe quais livros deveriam ser analisados, talvez não precisem saber que alguns textos estavam indisponíveis. Dar informações que os usuários não estejam esperandos pode reduzir a usabilidade de seu programa. As estruturas de tratamento de erros de Python permitem tem um controle minucioso sobre o que você deve compartilhar com os usuários quando algo sair errado; cabe a você decidir a quantidade de informações a serem compartilhadas.

Um código bem escrito, testado de modo apropriado, não será muito suscetível a erros internos, como erros de sintaxe ou de lógica. Contudo, sempre que seu programa depender de algo externo, por exemplo, de uma entrada de usuário, da existência de um arquivo ou da disponibilidade de uma conexão de rede, existe a possibilidade de uma exceção ser levantada. Um pouco de experiência ajudará você a saber em que pontos deverá incluir blocos de tratamento de exceções em seu programa e a quantidade de informações que deverá ser fornecida aos usuários sobre os erros que ocorrerem.

12.4 Armazenando dados

Quando os usuários fecham um programa, quase sempre você vai querer salvar as informações que eles forneceram. Uma maneira simples de fazer isso envolve armazenar seus dados usando o módulo `json`.

O módulo `json` permite descarregar estruturas de dados Python simples em um arquivo e carregar os dados desse arquivo na próxima vez que o programa executar. Também podemos usar `json` para compartilhar dados entre diferentes programas Python. Melhor ainda, o formato de dados JSON não é especifico de Python, portanto podemos compartilhar armazenados em formato JSON com pessoas que trabalhem com várias outras linguagens de programação. É um formato útil e portável, além de ser fácil de aprender.

O formato JSON (*JavaScript Object Notation*, ou Notação de Objetos JavaScript) foi originalmente desenvolvida para JavaScript. Apesar disso, tornou-se um formato comum, usado por muitas linguagens, incluindo Python.

12.4.1 Usando `json.dump()` e `json.load()`

Vamos escrever um pequeno programa que armazene um conjunto de números e outro que leia esses números de volta na memória. O primeiro programa usará `json.dump()` para armazenar o conjunto de números, e o segundo programa usará `json.load()`.

A função `json.dump()` aceita dois argumentos: um dado para armazenar e um objeto arquivo que pode ser usado para armazenar o dado. Eis o modo como podemos usar essa função para armazenar um lista de números.
Exemplo:

```
#Módulo
import json

#Lista a ser salva
numbers = [2,3,5,7,11,13]

#Caminho (Path) file
filename = "numbers.json"

#Escrever arquivo json
with open(filename,'w') as f_obj:
    json.dump(numbers, f_obj) #Salvar valores json
```

Inicialmente importamos o módulo `json` e criamos uma lista de números com o qual trabalharemos. Escolhemos o nome de um arquivo em que armazenaremos a lista de números. É comum

usar a extensão de arquivo `.json` para indicar que os dados do arquivo estão armazenados em formato JSON. Em seguida, abrimos o arquivo em **modo de escrita** ('w'), o que permite a `json` escrever os dados no arquivo. Usamos a função `json.dump()` para armazenar a lista `numbers` no arquivo `numbers.json`.

Esse programa não tem saída. Os dados estão armazenados em um formato que se parece com Python.

Agora escrevemos um programa que use `json.load()` para ler a lista de volta para a memória.

```
#Módulo
import json

#Arquivo a ser lido
filename = 'numbers.json'

#Abrir arquivo
with open(filename) as f_obj:
    numbers = json.load(f_obj) #Lendo arquivo json

print(numbers)
print(type(numbers))
```

Desta vez, quando abrimos o arquivo, fazemos isso em modo de leitura, pois o Python precisará apenas ler dados do arquivo. Usamos a função `json.load()` para carregar as informações armazenadas em `numbers.json` e as guardamos na variável `numbers`. Por fim, exibimos a lista de números recuperada. Podemos ver que é a mesma lista criada em `number_write.py`.

Essa é uma maneira simples de compartilhar dados entre dois programas.

Resumo:

1. Módulo `json`:

`Permite carregar e descarregar dados Python simples em um arquivo `.json`.`

2. `json.dump(dados,nome_arquivo.json)`:

`Armazena um conjunto de dados, formato de lista, no arquivo `.json`.`

3. `json.load(nome_arquivo.json)`:

`Carrega as informações armazenadas no arquivo `.json`.`

12.4.2 Salvando e lendo dados gerados pelo usuário

Salvar dados com `json` é conveniente quando trabalhamos com dados gerados pelo usuário porque, se você não salvar as informações de seus usuários de algum modo, elas serão perdidas quando o programa parar de executar. Vamos observar um exemplo em que pedimos aos usuários que forneçam seus nomes na primeira vez em que o programa executar e, então, o programa deverá lembrar esses nomes quando for executado novamente.

Vamos começar armazenando o nome do usuário:

```
#Bibliotecas
import json

#Entrada do usuário, nome
username = input("What is your name? ")

filename = "username.json" #Nome do arquivo

#Escrevendo arquivo json, uso da função json.dump()
with open(filename,'w') as f_obj:
    json.dump(username, f_obj) #Descarrega dados em um arquivo json
    print("We'll remember you when you come back, " + username + "!")
```

Pedimos o nome do usuário para que seja armazenado. Em seguida, usamos `json.dump()`, passando-lhe um nome de usuário e um objeto arquivo em que esse nome será armazenado. Então exebimos uma mensagem informando o usuário que armazenamos suas informações.

Vamos agora escrever um novo programa que faça uma saudação a um usuário cujo nome já esta armazenado.

```
#Biblioteca
import json #Biblioteca json

#Patch arquivo json
filename = "username.json"

with open(filename) as f_obj: #Abrindo arquivo
    username = json.load(f_obj) #Carrega informações armazenadas
    print("Welcome back, " + username + "!")
```

Usamos `json.load()` para ler as informações armazenadas em `username.json` na variável `username`. Agora que recuperamos o nome do usuário, podemos lhe desejar as boas-vindas de volta.

Precisamos combinar esses dois programas em um só arquivo. Quando alguém executar `remember_me.py`, queremos recuperar seu nome de usuário da memória, se for possível; assim, começaremos com um bloco `try`, que tentará recuperar o nome do usuário. Se o arquivo `username.json` não existir, faremos o bloco `except` pedir um nome de usuário e armazená-lo em username.json para ser usado da próxima vez.

```
import json

# Carrega o nome do usuário se foi armazenado anteriormente
# Caso contrário, pede que o usuário forneça o nome e armazena essa informação
filename = "username.json"

try:
    with open(filename) as f_obj:
        username = json.load(f_obj)
except FileNotFoundError:
    username = input("What is your name? ")
    with open(filename,'w') as f_obj:
        json.dump(username,f_obj)
        print("We'll remember you when you como back, "+
              username + "!")
else:
    print("Wellcome back, " + username + "!")
```

Não há nenhum código novo aqui; os blocos de código dos dois últimos exemplos simplesmente foram combinados em um só arquivo. Tentamos abrir o arquivo `username.json`. Se esse arquivo existir, lemos o nome do usuário de volta para a memória. e exibimos uma mensagem desejando boas-vindas de volta ao usuário no bloco `else`. Se essa é a primeira vez que o usuário executa o programa, `username.json` não existirá e um `FileNotFoundError` ocorrerá. O Python prosseguirá para o bloco `except`, em que pedimos ao usuário que forneça o seu nome. Então usamos `json.dump()` para armazenar o nome do usuário e exibimos uma saudação.

Qualquer que seja o bloco executado, o resultado será um nome de usuário e uma saudação apropriada.

12.4.3 Refatoração

Com frequência você chegará a um ponto em que seu código funcionará, mas reconhecerá que ele poderia ser melhorado se fosse dividido em uma série de funções com tarefas específicas. Esse processo se chama *refatoração*. A refatoração deixa seu código mais limpo, mais fácil de compreender e de estender.

Podemos refatorar `remember_me.py` passando a maior parte de sua lógica para uma ou mais funções. O foco de `remember_me.py` está na saudação ao usuário, portanto vamos transferir todo o código existente para uma função chamada `greet_user()`.

```
import json

def greet_user():
    """Saúda o usuário pelo nome."""
    # Carrega o nome do usuário se foi armazenado anteriormente
    # Caso contrário, pede que o usuário forneça o nome e armazena essa informação
    filename = "username.json"

    try:
        with open(filename) as f_obj:
            username = json.load(f_obj)
    except FileNotFoundError:
        username = input("What is your name? ")
        with open(filename,'w') as f_obj:
            json.dump(username,f_obj)
            print("We'll remember you when you como back, "+
                  username + "!")
    else:
        print("Wellcome back, " + username + "!")

greet_user()
```

Como estamos usando uma função agora, atualizamos os comentários com um docstring que reflete como o programa funciona no momento. Esse arquivo é um pouco mais limpo, porém a função `greet_user()` faz mais do que simplesmente saudar o usuário - ela também recupera um nome de usuário armazenado, caso haja um, e pede que o usuário forneça um novo nome, caso não exista.

Vamos refatorar `greet_user()` para que não faça tantas tarefas diferentes. começaremos transferindo o código para recuperar um nome de usuário já armazenado para uma função diferente.

```
import json

def get_stored_username():
    """Obtém o nome do usuário já armazenado se estiver disponível."""
    # Carrega o nome do usuário se foi armazenado anteriormente
    # Caso contrário, pede que o usuário forneça o nome e armazena essa informação
    filename = "username.json"

    try:
        with open(filename) as f_obj:
            username = json.load(f_obj)
    except FileNotFoundError:
        return None
    else:
        return username

def greet_user():
    """Saúda o usuário pelo nome."""
    username = get_stored_username()
    if username:
        print("Welcome back, " + username + "!")
    else:
        username = input("What is your name? ")
        filename = "username.json"
        with open(filename,'w') as f_obj:
            json.dump(username, f_obj)
            print("We'll remember you when you come back, " +
                  username + "!")

greet_user()
```

A nova função `get_stored_username()` tem um propósito claro, conforme infomado pela docstring. Essa função recupera um nome de usuário já armazenado e devolve esse nome se encontrar um. Se o arquivo `username.json` não existir, a função devolve `None`. Essa é uma boa prática: uma função deve devolver o valor esperado ou `None`. Isso nos permite fazer um teste simples com o valor de retorno da função. Exibimos uma mensagem de boas-vindas de volta ao usuário se a tentativa de recuperar um nome foi bem-sucedida; caso contrário, pedimos que um novo nome de usuário seja fornecido.

Devemos fatorar mais um bloco de código, removendo-o de `greet_user()`. Se o nome do usuário não existir, devemos transferir o código que pede um novo nome de usuário para uma função dedicada a esse propósito.

```
import json

def get_stored_username():
    """Obtém o nome do usuário já armazenado se estiver disponível."""
    # Carrega o nome do usuário se foi armazenado anteriormente
    # Caso contrário, pede que o usuário forneça o nome e armazena essa informação
    filename = "username.json"

    try:
        with open(filename) as f_obj:
            username = json.load(f_obj)
    except FileNotFoundError:
        return None
    else:
        return username

def get_new_username():
    """Pede um novo nome de usuário."""
    username = input("What is your name? ")
    filename = "username.json"
    with open(filename,'w') as f_obj:
        json.dump(username, f_obj)
    return username

def greet_user():
    """Saúda o usuário pelo nome."""
    username = get_stored_username()
    if username:
        print("Welcome back, " + username + "!")
    else:
        username = get_new_username()
        print("We'll remember you when you come back, " +
              username + "!")

greet_user()
```

Cada função nessa última versão de `remember_me.py` tem um propósito único e claro. Chamamos `greet_user()` e essa função exibe uma mensagem apropriada: ela dá as boas-vindas de volta a um usuário existente ou saúda um novo usuário. Isso é feito por meio da

chamada a `get_stored_username()`, que é responsável somente por recuperar um nome de usuário já armazenado, caso exista. Por fim, `greet_user()` chama `get_new_username()` se for necessário; essa função é responsável somente por obter um novo nome de usuário e armazená-lo. Essa separação do trabalho em compartimentos é uma parte essencial na escrita de um código claro, que seja fácil de manter e de estender.

13 Testando o código

Quando escrevemos uma função ou uma classe, podemos também escrever testes para esse código. Os testes provam que seu código funciona e deveria em resposta a todos os tipos de entrada para os quais ele foi projetado para receber. Ao escrever testes, você poderá estar confiante de que seu código funciona corretamente quando mais pessoas começarem a usar seus programas.

Você também poderá testar novos códigos a medida que adicioná-los para garantir que suas alterações não afetem o comportamento já existente em seu programa. Todo programador comete erros, portanto todo programador deve testar seus códigos com frequência, identificando os problemas antes que os usuários os encontrem.

Aprenderemos a testar o código usando ferramentas do módulo `unittest` de python. Veremos como criar um caso de teste e verificar se um conjunto de entradas resulta na saída desejada. Conheceremos a aparência de um teste que passa e de um teste que não passa, e veremos como um teste que falha pode nos ajudar a melhorar o código.

Aprenderemos a testar funções e classes, e você começará a entender quantos testes devem ser escritos para um projeto.

13.1 Testando uma função

Para aprender a testar, precisamos de um código para testes. Eis uma função simples que aceita um primeiro nome e um sobrenome e devolve um nome completo formatado de modo elegante.

```
def get_formatted_name(first,last):
    """Gera um nome completo formatado de modo elegante."""
    full_name = first + " " + last
    return full_name.title()
```

A função `get_formatted_name()` combina o primeiro nome e o sobrenome com um espaço entre eles para compor um nome completo e então converte as primeiras letras do nome para maiúsculas e devolve o nome completo.

Para verificar se `get_formatted_name()` funciona, vemos criar um programa que use essa função. O programa `names.py` permite que os usuários forneçam um primeiro nome e um sobrenome e vejam um nome completo formatado de modo elegante.

```
from name_function import get_formatted_name  # Importa função

print("Enter 'q' at any time to quit.")  # Mensagem para sair
while True:  # Loop principal
    first = input("\nPlase give me a first name: ")  # Pede primeiro nome
    if first == 'q':  # Sai se digitar 'q'
        break
    last = input("\nPlase give me a last name: ")  # Pede último nome
    if last == 'q':  # Sai se digitar 'q'
        break

    formatted_name = get_formatted_name(first, last)  # Formata o nome
    print("\tNeatly formatted name : " + formatted_name + ".")
    # Imprime nome formatado
```

Esse programa importa `get_formatted_name()` de `name_function.py`. O usuário pode fornecer uma série de primeiros nomes e de sobrenomes e ver os nomes completos formatados.

Podemos observar que os nomes gerados nesse caso estão corretos. Porém, vamos supor que queremos modificar `get_formatted_name()` para que ele seja capaz de lidar com nomes do meio também. Quando fizermos isso, queremos ter certeza de que não causamos erros no modo como a função trata os nomes que tenham apenas primeiro nome e sobrenome. Poderiamos testar nosso código executando `names.py` e fornecendo um nome como `Janis Joplin` sempre que modificarmos `get_formatted_name()`, mas isso seria tedioso. Felizmente o Python oferece um modo eficiente de automatizar os testes da saída de uma função. Se os testes de `get_formatted_name()` forem automatizados, poderemos sempre ter a confiança de que a função estará correta quando fornecemos os tipos de nomes para os quais os testes forem escritos.

13.1.1 Testes de unidade e casos de teste

O módulo `unittest` da biblioteca-padrão de Python oferece as ferramentas para testar seu código.

Conceitos básicos de teste:

- Um **teste de unidade** verifica se um aspecto específico do comportamento de uma função está correto.
- Um **caso de teste** é uma coleção de testes de unidade que, em conjunto, prova que uma função se comporta como deveria em todas as situações que você espera que ela trate. Um bom caso de teste considera todos os tipos possíveis de entradas que uma função poderia receber e inclui testes para representar cada uma dessas situações.
- Um caso de teste com **cobertura completa** é composto de uma variedade de testes de unidade que inclui todas as possíveis maneiras de usar uma função. Atingir a cobertura completa em um projeto de grande porte pode ser desanimador. Em geral, é suficiente escrever testes para os comportamentos críticos de seu código e então visar uma cobertura completa somente se o projeto começar a ter uso disseminado.

Resumo:

- *Teste de unidade*:

  Verifica se um aspecto específico do comportamento de uma função esta correto.

- *Caso de teste*:

  É uma coleção de testes de unidade que, em conjunto, prova que uma função se comporta como deveria em todas as situações que você espera que ela trate.

- Caso de teste com *Cobertura completa*:

  É composto de uma variedade de de testes de unidade que inclui todas as possiveis maneiras de usar uma função.

13.1.2 Um teste que passa

A sintaxe para criar um caso de teste exige um pouco de prática, mas depois que você o configura, será mais fácil adicionar outros casos de teste para suas funções. Para escrever um caso de teste para uma função, importe o módulo `unittest` e a função que você que testar. Em seguida crie uma classe que herde de `unittest.TestCase` e escreva uma série de métodos para testar diferentes aspectos do comportamento de sua função.

Eis um caso de teste com um método que verifica se a função `get_formatted_name()` esta correta quando recebe um primeiro nome e um sobrenome:

```
import unittest  # Importa biblioteca de testes
from name_function import get_formatted_name  # Importa função

class NameTestCase(unittest.TestCase):  # Cria classe de teste
  """Testes para 'name_function.py'"""  # Docstring da classe de teste

  def test_first_last_name(self):  # Define um método de teste
    """Nomes como 'janis joplin' funcionam?"""  # Docstring do método de teste
    formatted_name = get_formatted_name('janis', 'joplin')  # Chama função para formatar nome
    self.assertEqual(formatted_name, 'Janis Joplin')  # Verifica se o nome formatado está co

unittest.main()  # Executa os testes
```

- Inicialmente importamos `unittest` e a função `get_formatted_name()` que queremos testar.
- Criamos uma classe chamada `NamesTestCase`, que conterá uma série de testes de unidade para `get_formatted_name()`. Você pode dar o nome que quiser para a classe, mas é melhor nomea-lá com palavras relacionadas a função que você está prestes a testar e usar a palavra *Test* no nome da classe. Essa classe deve herdar da classe `unittest.TestCase` para que o Python saiba executar os testes que você escrever.

  `NamesTestCase` contém um único método que testa um aspecto de `get_formatted_name()`. Chamamos esse método de `test_first_last_name()` porque estamos verificando se os nomes que têm apenas o primeiro nome e o sobrenome são formados corretamente. Qualquer método que comece com `test_` serpa executado de modo automático quando `test_name_function.py` for executado. Nesse método de teste, chamamos a função que queremos testar e armazenamos um valor de retorno que estamos interessados em testar. Nesse exemplo `get_formatted_name()` com os argumentos 'janis' e 'joplin' e armazenamos o resultado em `formatted_name`.

- Método de *asserção*:

  Usamos um dos recursos mais úteis de `unittest`: um método de *asserção*.

  Os métodos de asserção verificam se um resultado recebido é igual ao resultado que você espera receber.

  Nesse caso, como sabemos que `get_formatted_name()` deve devolver um nome complexo, com as letras iniciais maiúsculas e os espaços apropriados, esperamos que o valor em `formatted_name` seja 'Janis Joplin'. Para conferir se isso é verdadeiro, usamos o método `assertEqual()` de `unittest` e lhe passamos `formatted_name` e 'Janis Joplin'.

  A linha:

  ```
  self.assertEqual(formatted_name, 'Janis Joplin')
  ```

  diz o seguinte:
  "Compare o valor em `formatted_name` com a string 'Janis Joplin'. Se forem iguais conforme esperado, tudo bem. Contudo, se não forem iguais, me avise!"

- A linha `unittest.main()` diz para o Python executar os testes desse arquivo.

  Quando executamos `test_name_function.py`, vemos a saida a seguir:

  ```
  .
  ----------------------------------------------------------------------
  Ran 1 test in 0.001s

  OK
  ```

  - O ponto na primeira linha de saída nos informa que um único teste passou.
  - A próxima linha diz que o Python executou 1 teste e durou menos de $0,001$ segundos para faze-lo.
  - O 'OK' no final informa que todos os testes de unidade do caso de teste passaram.

  Essa saída mostra que a função `get_formatted_name()` sempre funcionará para nomes que tenham o primeiro nome e o sobrenome, a menos que a função seja modificada. Se modificarmos `get_formatted_name()`, poderemos executar esse teste novamente. Se o caso de teste passar, saberemos que a função continua funcionando para nomes como 'Janis Joplin'.

13.1.3 Um teste que falha

Como é a aparência de um teste que falha? Vamos modificar `get_formatted_name()` para que possa tratar nomes do meio, mas faremos isso de modo que a função gere um erro para nomes que tenham apenas um primeiro nome e um sobrenome, como 'Janis Joplin'.

A seguir, apresentamos uma nova versão de `get_formatted_name()` que exige um argumento para um nome do meio:

```
def get_formatted_name(first, middle, last):
    """Gera um nome completo formatado de modo elegante.

    Args:
        first (str): O primeiro nome.
        middle (str): O nome do meio.
        last (str): O sobrenome.

    Returns:
        str: O nome completo formatado.
    """

    # Concatena as partes do nome com espaços
    full_name = first + " " + middle + " " + last

    # Formata o nome para capitalizar a primeira letra de cada palavra
    return full_name.title()
```

Essa versão deve funcionar para pessoas com nomes do meio, mas quando a testamos, vemos que ela deixou de funcionar para pessoas que tenham apenas um primeiro nome e um sobrenome.

Dessa vez, a execução do arquivo `test_name_function.py` fornece o resultado a seguir:

```
E
======================================================================
ERROR: test_first_last_name (__main__.NameTestCase.test_first_last_name)
Nomes como 'janis joplin' funcionam?
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/sergio/Programacao/python/Cap11-Testando_codigo/01-Testando_funcao/test_name_fu
    formatted_name = get_formatted_name('janis', 'joplin')  # Chama função para formatar nome
                     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
TypeError: get_formatted_name() missing 1 required positional argument: 'last'

----------------------------------------------------------------------
Ran 1 test in 0.003s

FAILED (errors=1)
```

Há muitas informações aqui, pois há muitos dados que você precisa saber quando um teste falha:

- O primeiro item da saída é um único `E`, que nos informa que um teste de unidade do caso de teste resultou em erro.
- A seguir, vemos que `test_first_last_name()` em `NamesTestCase` causou um erro. Saber qual teste falhou será crucial quando o seu caso de teste tiver muitos testes de unidade.
- Um tracenack padrão, que informa que a chamada de função `get_formatted_name('janis','joplin')` não funciona mais, pois um argumento posicional obrigatório está ausente.
- Também podemos ver que um único teste de unidade foi executado.
- Por fim, vemos uma mensagem adicional informando que o caso de teste como um todo falhou e que houve um erro em sua execução. Essa informação aparece no final da saída para que possa ser vista de imediato; você não vai querer fazer uma rolagem para cima em uma listagem longa de saída para descobrir quantos testes falharem.

13.1.4 Respondendo a um teste que falhou

O que devemos fazer quando um teste falha? Supondo que você esteja verificando as condições corretas,um teste que passa significa que a função esta se comportando de forma apropriada e um teste que falha quer dizer que há um erro no novo código que você escreveu. Assim, **se um teste falhar, não mude o teste. Em vez disso, corrija o código que fez o teste falhar**. Analise as alterações que você acabou de fazer na função e descubra como elas afetaram o comportamento desejado.

Exemplo:

Nesse caso, `get_formatted_name()` costumava exigir apenas dois parâmetros: um primeiro nome e um sobrenome. Agora ele exige um primeiro nome, um nome do meio e um sobrenome. A adição do parâmetro obrigatório para o nome do meio fez o comportamento desejado de `get_formatted_name()` apresentar problemas. A melhor opção nesse caso é deixar o nome do meio opcional. Feito isso, nosso teste 'Janis Joplin' deverá passar novamente e poderemos aceitar nomes do meio também. Vamos modificar `get_formatted_name()` de modo que os nomes do meio sejam opcionais e então executar o caso de teste novamente. Se o teste passar, prosseguiremos para garantir que a função trate os nomes do meio de forma apropriada.

Para deixar os nomes do meio opcionais, passamos o parâmetro `middle` para o final da lista de parâmetros na definição da função e lhe oferecemos um valor default vazio. Além disso, acrescentamos um teste `if` que compõe o nome completo de forma apropriada, comforme um nome do meio tenha sido fornecido ou não.

```
def get_formatted_name(first, last, middle=''):
    """Gera um nome completo formatado de modo elegante.

    Args:
        first (str): O primeiro nome.
        middle (str): O nome do meio (opcional).
        last (str): O sobrenome.

    Returns:
        str: O nome completo formatado.
    """
    if middle:
        # Concatena as partes do nome com espaços, incluindo o nome do meio
        full_name = first + " " + middle + " " + last
    else:
        # Concatena as partes do nome com espaços, sem o nome do meio
        full_name = first + " " + last

    # Formata o nome para capitalizar a primeira letra de cada palavra
```

```
    return full_name.title()
```

Nessa nova versão de `get_formatted_name()`, o nome do meio é opcional. Se um nome do meio for passado para a função (`if middle:`), o nome completo conterá um primeiro nome, um nome do meio e um sobrenome. Caso contrário, o nome completo será constituído apenas de um primeiro nome e de um sobrenome. Agora a função deve estar adequada para trabalhar com os dois tipos de nomes. Para descobrir se a função continua apropriada para nomes como 'Janis Joplin', vamos executar `test_name_function.py` novamente.

```
.
----------------------------------------------------------------------
Ran 1 tests in 0.000s

OK
```

O caso de teste agora passou. É a situação ideal: quer dizer que a função está correta para nomes como 'Janis Joplin' de novo, sem que tenhamos que testar a função manualmente. Corrigir nossa função foi fácil porque o teste que falhou nos ajudou a identificar o novo código que interferiu no comportamento existente.

13.1.5 Adicionando novos testes

Agora que sabemos que `get_formatted_name()` funciona para nomes simples novamente, vamos escrever um segundo teste para pessoas que tenham um nome do meio. Fazemos isso adicionando outro método à classe `NamesTestCase`:

```
import unittest  # Importa a biblioteca unittest para testes

from name_function import get_formatted_name  # Importa a função que será testada

class NameTestCase(unittest.TestCase):  # Define uma classe para os casos de teste, herdando
    """Testes para 'name_function.py'"""  # Docstring da classe, descrevendo o propósito dos

    def test_first_last_name(self):  # Define um método de teste para nomes com primeiro e úl
        """Nomes como 'janis joplin' funcionam?"""  # Docstring do método, explicando o caso
        formatted_name = get_formatted_name('janis', 'joplin')  # Chama a função com um nome
        self.assertEqual(formatted_name, 'Janis Joplin')  # Verifica se o resultado formatado

    def test_first_last_middle_name(self):  # Define um método de teste para nomes com primei
        """Nomes como 'Wolfgang Amadeus Mozart' funcionam?"""  # Docstring do método, explica
        formatted_name = get_formatted_name('wolfgang', 'mozart', 'amadeus')  # Chama a funçã
        self.assertEqual(formatted_name, 'Wolfgang Amadeus Mozart')  # Verifica se o resultad

unittest.main()  # Executa os testes definidos na classe
```

Chamamos esse novo método de `test_first_last_middle_name()`. O nome do método deve começar com `test_` para que seja executado automáticamente quando `test_name_function.py` for executado. Nomeamos o método de modo a deixar claro qual é o comportamento de `get_formatted_name()` que estamos testando. Como resultado, se o teste falhar, saberemos de imediato quais tipos de nome serão afetados.

Nomes longos para os métodos em nossas classe `TestCase` não são um problema. Eles devem ser descritivos para que você possa compreender a saída quando seus testes falharem, e pelo fato do Python os chamar automaticamente, você não precisará escrever código para chamar esses métodos.

Para testar a função, chamamos `get_formatted_name()` com um primeiro nome, um sobrenome e um nome do meio e, em seguida, usamos `assertEqual()` para conferir se o nome completo devolvido coincide com o nome completo (primeiro nome, nome do meio e sobrenome) esperado. Se executarmos `test_name_function.py` novamente, veremos que os dois testes passam:

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.003s

OK
```

13.2 Testando uma classe

13.2.1 Uma variedade de métodos de asserção

13.2.2 Uma classe para testar

13.2.3 Testando a classe `AnonymousSurvey`

13.2.4 Método `setUp()`

Bibliografia

[1] Eric Matthes. *Curso Intensivo de Python - 3ª Edição: Uma Introdução Prática e Baseada em Projetos à Programação*. Novatec Editora, 2023. ISBN: 9788575228432. URL: https://books.google.com.br/books?id=mkW7EAAAQBAJ.