ENGENHARIA DE *SOFTWARE* PARA INTELIGÊNCIA ARTIFICIAL

UNIDADE II

COMEÇANDO A PROGRAMAR

Elaboração

Diogo Santos Silva da Costa

Atualização

Alexander Francisco Vargas Salgado

Produção

Equipe Técnica de Avaliação, Revisão Linguística e Editoração

SUMÁRIO

CAPÍTULO 1

LINGUAGEM DE PROGRAMAÇÃO

O que são linguagens de programação?

As linguagens de programação de computadores são ferramentas que nos permitem inserir instruções para um computador de forma que estes as entendam. Do mesmo modo que existem diversas linguagens de comunicação humana, existem também as de computadores. A parte que o computador consegue entender nessa comunicação é a linguagem binária, e o processo de tradução do que escrevemos para essa linguagem é chamado de compilação. As linguagens permitem que computadores realizem diversas tarefas de manipulação de dados de forma bem mais eficiente que um ser humano faria, por exemplo.

Python

Python é uma linguagem de programação estruturada, orientada a objeto, de alto nível e interpretada. Possui um modelo de desenvolvimento baseado em comunidades e fóruns, sendo aberto e gratuito. É uma linguagem que prioriza a legibilidade do código e possui inúmeras bibliotecas disponíveis na internet, além de ser uma das mais utilizadas no mundo em diversas aplicações, inclusive na Inteligência Artificial.

Python possui baixo uso de caracteres especiais, fazendo com que seja muito parecida com pseudo-código-executável. Ademais, faz uso de indentação para marcar blocos, possui coletor de lixo para gerenciar automaticamente o uso da memória, entre tantas outras características. Uma outra grande vantagem de Python é ser multiplataforma, ou seja, pode ser migrada para outras plataformas sem nenhum tipo de alteração no código escrito.

É considerada uma linguagem de propósito geral, capaz de lidar com todo tipo de tarefa: buscar dados em bancos de dados, ler páginas na internet, exibir resultados em

formato gráfico, criar planilhas, etc. Já as linguagens científicas normalmente são muito específicas em seus propósitos.

Instalação do Python

Para a instalação do *Python* em seu computador, acesse o *link* oficial https://www.python.org/. Nesse endereço, estão contidos todos os arquivos e os passos necessários para que você possa ter esse interpretador instalado em sua máquina.

Além do interpretador *Python*, é importante ter o programa *Pip*. Este é um sistema que gerencia pacotes na linguagem *Python*, para serem utilizados em seus programas, popularmente conhecidos como bibliotecas. As informações sobre instalação e como usar podem ser obtidas em https://pypi.org/project/pip/.

Por que usar Python?

Python é uma linguagem de alto nível, por isso não é necessário conhecer detalhes do *hardware* que está sendo utilizado. Além disso, a escrita em python é altamente simplificada, oferecendo estruturas simples, mas poderosas, eliminando redundâncias, o que torna o código legível e organizado como critério para um bom funcionamento. Pela sua simplicidade, Python é bem fácil e rápido de se aprender, mesmo sem conhecimento anterior em outras linguagens de programação.

Essa linguagem possui centenas de milhares de módulos disponíveis gratuitamente na *internet*, que todos podem utilizar, além de uma comunidade crescente e aberta, o que permite tirar dúvidas com muita facilidade. Ter módulos e bibliotecas disponíveis é uma das grandes vantagens de qualquer ferramenta, isso torna as tarefas mais fáceis, uma vez que é fácil encontrar códigos e pacotes que já atendem às demandas que alguém, assim como você, buscou. Além disso, se trata de uma linguagem de programação multiplataforma, isso significa que um programa escrito em *Python* pode ser utilizado em diversos sistemas operacionais e arquiteturas com poucas modificações, ou até mesmo nenhuma.

Mas o principal motivo da adoção de Python nestes estudos é a adoção da linguagem pelas comunidades de Ciência de Dados, Inteligência Artificial e *Big Data*. Python, assim como R e Scala, recebe implementações das principais ferramentas do ramo, além de haver códigos de exemplo e materiais de referência acessíveis.

Codificando em Python

Você pode nunca ter tido contato com uma lingagem de programação ou pode estar acostumado com outras linguagens, talvez linguagens em que seja comum o uso de delimitadores, como o ponto e vírgula. Em *Python*, isso não acontece, o interpretador separa linha a linha e é baseado na identação, em vez de delimitadores como em linguagens Java ou C. A indentação em *Python* vai além da estética da legibilidade em um programa, ela faz parte da sintaxe.

Espaços em branco no início da linha não são ignorados, são entendidos como organização do código dentro de um bloco de comandos. Porém, espaços em branco dentro de parênteses e colchetes não possuem relevância dentro do programa.

Os exemplos aqui presentes são baseados em sistema operacional Linux, que é o sistema mais comum em *softwares* do tipo. Em geral, é possível lidar com outros sistemas operacionais sem grandes dificuldades, mas pode ser interessante usar um sistema de virtualização como Docker ou uma máquina virtual.

Existem algumas formas de acessar o interpretador Python, a mais fácil é usando o *REPL* em um terminal de texto puro. Outra é usando o jupyter, o qual fornece um REPL em interface gráfica. Por último, está a criação de *scritps*: nesse caso, abra um editor de texto puro ou um editor de códigos e insira seu código para ser executado posteriormente. Ao longo deste conteúdo, vamos usar os modelos interativos, que podem ser empregados no terminal ou no jupyter (didaticamente, recomendamos o jupyter). Para aprender a acessar essas ferramentas, veja o material complementar.

Primeiro programa em Python

Antes de partir para a manipulação de dados, vamos apresentar um pouco mais da linguagem Python. Para tanto, criaremos o programa clássico de introdução a qualquer linguagem, uma versão do "*Olá, Mundo*", que apenas irá exibir na tela do computador um texto. Para isso, podemos escrever o seguinte código:

```
1 # Este é nosso primeiro programa em python
2 print("Olá Mundo")
```

```
Olá Mundo
```

Analisando um pouco mais a fundo:

» a primeira linha não dará saída alguma para a tela. Repare que ela se inicia com o caractere cerquilha (#). Esse caractere é especial e faz com que todo o conteúdo escrito a partir dele seja ignorado, chamamos isso de comentário. Comentários são linhas de código usadas para documentação, você as verá com frequência, já que aqui apresentamos códigos fundamentalmente didáticos;

- a segunda linha utiliza uma função básica, `print`. O argumento dessa função é impresso na saída primária, geralmente a tela do computador;
- outro aprendizado é que, como é comum em outras linguagens, textos em *Python* devem estar escritos entre aspas simples ou duplas.

Ao final do programa, teremos como resultado:

```
1 # Este é nosso primeiro programa em python
2 print("Olá Mundo")
```

```
Olá Mundo
```

Existem algumas formas de executar códigos Python, elas se dividem entre usar a interface interativa, o REPL, seja no terminal, seja em alguma interface como o iPython ou o Jupyter, ou criar uma aplicação, nesse caso o código é salvo num arquivo em formato .py.

Ao longo deste estudo, vamos intercambiar essas técnicas quando for mais adequado.

Os sistemas interativos são mais usados para experimentos ou tarefas rápidas. À medida que aprendemos uma nova ferramenta, preparamos um conjunto de dados para alimentar um modelo de aprendizado de máquina ou fazemos análise de dados a fim de gerar um relatório, a interface interativa se torna mais interessante. No entanto, quando temos uma aplicação que será disponibilizada como um serviço, ou mesmo que precisa ser executada novamente, criamos um módulo no formato .py.

CAPÍTULO 2

TIPOS DE DADOS E ESTRUTURA DE DADOS

Conhecendo os tipos de dados

Linguagens de programação têm o objetivo fundamental de manipular dados e reagir a eventos. Nesta etapa, vamos apresentar os tipos básicos de dados que podem ser manipulados em Python. Os tipos mais básicos são texto, *string* e tipos numéricos. Em Python, estes são as strings, do tipo str, os inteiros do tipo int e os números de ponto flutuante.

```
logradouro = 'Avenida Monte AlegreRio Claro'
numero = 27
bairro = 'Jardins'
municipio = 'Vila Josefa'
estado = 'Tocantins'
endereco = logradouro + str(numero) + ',' + bairro + ' - ' + municipio + ',' + estado
print(endereco)
```

```
Avenida Monte AlegreRio Claro27,Jardins - Vila Josefa,Tocantins
```

A saída desse código será como mostrado acima.

Nesse código, aprendemos algumas questões sobre *strings*:

- são delimitadas por aspas duplas (“), mas também podem ser delimitadas por aspas simples (‘);
- podem ser concatenadas com outras *strings* usando o operador +;
- inteiros (e números em geral) podem ser convertidos para *string* sendo usados como argumento do construtor da classe `str`.

Quanto a números, você já deve imaginar o resultado esperado:

```
x = 10
y = 2.3
print(x+y)
print(y*x)
print(x/3+y*2)
```

```
12.3
23.0
7.933333333333334
```

Vamos analisar este código:

`x + y nada mais é do que a soma dos valores de x e y;`

`y*x é a multiplicação de x por y;`

`x/3 é a divisão de x por três.`

Isso foi bem óbvio, mas veremos casos um pouco menos óbvios da operação com *strings* e números.

Outro grupo de tipos são as coleções, e o primeiro representante das coleções são as listas, que é um tipo básico. Listas são coleções ordenadas de elementos que podem ser acessados por meio de índices inteiros indicando sua posição. Por exemplo, façamos listas de estados brasileiros:

```
sudeste = ['RJ', 'SP', 'ES', 'MG']
sul = ['PR', 'SC', 'RS']
```

Podemos concatenar as duas listas com o operador +;

```
estados = sudeste + sul
print(estados)
```

```
['RJ', 'SP', 'ES', 'MG', 'PR', 'SC', 'RS']
```

E acessar elementos pelo índice:

```
estados[0]
```

```
'RJ'
```

Podemos adicionar um novo elemento ao final da lista:

```
estados.append('TI')
```

Resultando em:

```
print(estados)
```

```
['RJ', 'SP', 'ES', 'MG', 'PR', 'SC', 'RS', 'TI']
```

Podemos ver o tamanho da lista:

```
len(estados)
```

```
8
```

E modificar um elemento a partir do índice:

```
estados[7] = 'TO'
```

Transformando a lista em:

```
print(estados)
```

```
['RJ', 'SP', 'ES', 'MG', 'PR', 'SC', 'RS', 'TO']
```

Para além disso, tuplas são como listas, mas são imutáveis. Uma tupla com os estados do sudeste seria criada por:

```
sudeste = ('RJ', 'SP', 'ES', 'MG')
```

Com os estados do sul:

```
1 sul = ('PR', 'SC', 'RS')
```

E as listas podem novamente ser concatenadas pelo operador +, mas uma tupla não pode ser modificada, isto é:

```
1 sudeste[0] = 'RI'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-27-2ddcf03d8f70> in <module>
----> 1 sudeste[0] = 'RI'

TypeError: 'tuple' object does not support item assignment
```

Ocorre um erro de execução, como pode ser visto acima.

Uma outra coleção é os conjuntos. Conjuntos são coleções não ordenadas, não podemos acessar elementos por índices e, assim como os conjuntos da matemática, os elementos não se repetem.

Podemos criar um conjunto a partir de uma lista ou tupla:

```
1 estados = set(['RJ', 'SP', 'ES', 'SP'])
```

Isso resulta na eliminação das repetições:

```
1 estados = {'RJ', 'SP', 'ES'}
```

Podemos adicionar novos elementos a um conjunto usando o método `.add`.

```
1 estados.add('SC')
```

```
1 print(estados)
```

```
{'ES', 'SP', 'RJ', 'SC'}
```

E mesclar conjuntos usando o método `update`:

```
1 sudeste = set(['RJ', 'SP'])
2 outros = set(['RJ', 'SC'])
3 outros.update(sudeste)
4 print(outros)
```

```
{'SP', 'RJ', 'SC'}
```

A última estrutura são os dicionários. Dicionários são mapas de chave-valor, permitem-nos definir a relação entre seus elementos:

```
1 sudeste = {'RJ':21, 'SP':11, 'MG':31, 'ES':27}
2 prefixo_RJ = ['RJ']
3 print(prefixo_RJ)
```

```
['RJ']
```

Se tentarmos utilizar uma chave que não exista no dicionário, este irá retornar com um erro, mas temos a possibilidade de verificar se aquela chave é existente ou não. Veja a seguir:

```
False
```

```
prefixo_RN = sudeste['RN']
prefixo_tem_RN = 'RN' in sudeste
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-37-9d73f4706b7b> in <module>
----> 1 prefixo_RN = sudeste['RN']
      2 prefixo_tem_RN = 'RN' in sudeste

KeyError: 'RN'
```

```
True
```

```
prefixo_RN = sudeste['RJ']
prefixo_tem_RN = 'RJ' in sudeste
print(prefixo_tem_RN)
```

```
True
```

Dicionários também possuem um método `.get` que retorna um valor no lugar de erro quando você procura por uma chave inexistente. Assim podemos ver abaixo o uso do `.get`:

```
RN_prefixo = sudeste.get('RN',0)
print (RN_prefixo)
```

```
0
```

```
RJ_prefixo = sudeste.get('RJ',0)
print (RJ_prefixo)
```

```
21
```

Podemos atribuir atualizações nos valores-chave dos pares da seguinte forma:

```
sudeste['SP'] = 12
sudeste['RJ'] = 22
sudeste['MG'] = 32
sudeste['ES'] = 28
print (sudeste)
```

```
{'RJ': 22, 'SP': 12, 'MG': 32, 'ES': 28}
```

Os dicionários são bastante úteis na representação de dados estruturados, podemos empregar alguns métodos como `.keys` para obter as chaves, `.values` para obter os valores associados nas chaves e `.items`, que lista os itens contidos no dicionário. Observe:

```
usuario = {'nome':'Pedro', 'id': 'pedro_user', 'email':'pedro@email'}
print (usuario)
```

```
{'nome': 'Pedro', 'id': 'pedro_user', 'email': 'pedro@email'}
```

```
usuario_keys = usuario.keys()
usuario_values = usuario.values()
usuario_items = usuario.items()
print (usuario_keys)
print (usuario_values)
print (usuario_items)
```

```
dict_keys(['nome', 'id', 'email'])
dict_values(['Pedro', 'pedro_user', 'pedro@email'])
dict_items([('nome', 'Pedro'), ('id', 'pedro_user'), ('email', 'pedro@email')])
```

Estruturas condicionais

Na forma mais básica, o programa pode ser entendido como um *pipeline* de dados, um fluxo que indica a próxima instrução a ser executada.

Por exemplo, supomos uma aplicação que processa um conjunto de dados referente a residentes no estado de SC, mas existe a possibilidade de um usuário cadastrar elementos de um estado diferente. Tornando o exemplo mais claro, veja o esquema a seguir.

Recebemos um dicionário no seguinte formato:

```
investidor1 = {'nome': 'José Ramalho', 'estado': 'SC', 'renda anual': 80000.00}
print (investidor1)
investidor2 = {'nome': 'Roberto Silva', 'estado': 'RJ','renda anual': 60000.00}
print (investidor2)
```

```
{'nome': 'José Ramalho', 'estado': 'SC', 'renda anual': 80000.0}
{'nome': 'Roberto Silva', 'estado': 'RJ', 'renda anual': 60000.0}
```

Esse sistema transforma os dados para calcular a renda real por ano de uma aplicação, tais dados serão usados como entrada para um modelo de aprendizado. O sistema usa a renda anual aplicando uma taxa de 23% para determinar o valor final a receber; mas, por questões fiscais, os moradores de SC têm uma taxa de 15%, de forma que nosso normalizador deve refletir isso. Nessa etapa, usamos uma condicional, que, em python, é a estrutura `if`.

Vejamos o exemplo a seguir para os investidores um e dois:

```
if investidor1['estado'] == 'SC':
 investidor1['renda final'] = investidor1['renda anual']*(1.0-0.15)
else:
 investidor1['renda final'] = investidor1['renda anual']*(1.0-0.23)
investidor1
{'nome': 'José Ramalho',
 'estado': 'SC',
 'renda anual': 80000.0,
 'renda final': 68000.0}
if investidor2['estado'] == 'SC':
 investidor2['renda final'] = investidor2['renda anual']*(1.0-0.15)
else:
 investidor2['renda final'] = investidor2['renda anual']*(1.0-0.23)
investidor2
{'nome': 'Roberto Silva',
 'estado': 'RJ',
 'renda anual': 60000.0,
 'renda final': 46200.0}
```

```
{'nome': 'Roberto Silva',
 'estado': 'RJ',
 'renda anual': 60000.0,
 'renda final': 46200.0}
```

Nesse trecho de código, temos duas diferentes possibilidades: se o estado for ‘SC’, o sistema entra na primeira condição, se o estado for qualquer outro, o sistema entra na segunda condição.

Se o problema for um pouco mais complexo e houver outros estados com a taxa, teremos uma estrutura um pouco mais complexa, mas ainda muito simples. Suponhamos que os investidores do RJ tenham uma taxa de 17%.

Veremos essa questão para o investidor um e para o investidor dois:

```
if investidor1['estado'] == 'SC':
 investidor1['renda final'] = investidor1['renda anual']*(1.0-0.15)
elif investidor1['estado'] == 'RJ':
 investidor1['renda final'] = investidor1['renda anual']*(1.0-0.17)
else:
 investidor1['renda final'] = investidor1['renda anual']*(1.0-0.23)
investidor1
{'nome': 'José Ramalho',
 'estado': 'SC',
 'renda anual': 80000.0,
 'renda final': 68000.0}
if investidor2['estado'] == 'SC':
 investidor2['renda final'] = investidor2['renda anual']*(1.0-0.15)
elif investidor2['estado'] == 'RJ':
 investidor2['renda final'] = investidor2['renda anual']*(1.0-0.17)
else:
 investidor2['renda final'] = investidor2['renda anual']*(1.0-0.23)
investidor2
{'nome': 'Roberto Silva',
 'estado': 'RJ',
 'renda anual': 60000.0,
 'renda final': 49800.0}
```

```
{'nome': 'Roberto Silva',
 'estado': 'RJ',
 'renda anual': 60000.0,
 'renda final': 49800.0}
```

Nesse cenário, se o estado for SC, a primeira instrução é executada; se não for SC, mas sim RJ, a segunda instrução é executada; e, não sendo nem SC nem RJ, a terceira instrução é executada. Por fim, temos um último caso, em que as taxas são aplicadas apenas para RJ e SC, mas nenhum outro estado aplica taxas na renda anual do investidor. Dessa forma, podemos omitir a instrução *else*.

```
if investidor1['estado'] == 'SC':
 investidor1['renda final'] = investidor1['renda anual']*(1.0-0.15)
elif investidor1['estado'] == 'RJ':
 investidor1['renda final'] = investidor1['renda anual']*(1.0-0.17)
investidor1
{'nome': 'José Ramalho',
 'estado': 'SC',
 'renda anual': 80000.0,
 'renda final': 68000.0}
if investidor2['estado'] == 'SC':
 investidor2['renda final'] = investidor2['renda anual']*(1.0-0.15)
elif investidor2['estado'] == 'RJ':
 investidor2['renda final'] = investidor2['renda anual']*(1.0-0.17)
investidor2
{'nome': 'Roberto Silva',
 'estado': 'RJ',
 'renda anual': 60000.0,
 'renda final': 49800.0}
```

```
{'nome': 'Roberto Silva',
 'estado': 'RJ',
 'renda anual': 60000.0,
 'renda final': 49800.0}
```

As estruturas condicionais são blocos de comandos que são executados quando uma condição verificada é verdadeira. Caso essa condição não seja verdadeira, outra condição pode ser testada, ou até mesmo podemos encerrar todo o bloco de ações.

» Nesta primeira estrutura, testamos apenas uma condição. Caso não seja verdadeira, o bloco é encerrado:

```
if <condição verificada >:
Ação executada
```

» Nesta outra, testamos uma condição que, caso seja verdadeira, executa o bloco Ação 1, do contrário, o bloco Ação 2:

```
if <condição verificada >:
Ação 1 executada
else:
Ação 2 executada
```

Podemos ter um caso em que mais de uma condição é verificada. Porém, se todas essas condições retornarem valores falsos, todo o conjunto condicional é encerrado sem que nada seja feito:

```
if <condição 1 verificada >:
Ação 1 executada
elif <condição 2 verificada >:
Ação 2 executada
elif <condição 3 verificada >:
Ação 3 executada
```

Por fim, podemos testar mais de uma condição usando *elif* quantas vezes forem necessárias. No caso de a condição 1 ser verdadeira, a Ação 1 é executada. Se o primeiro teste for negativo, a condição 2 é testada para possivelmente executar a Ação 2. Essa estrutura pode ser repetida quantas vezes forem necessárias. Caso todos os testes gerem resultados negativos, o último bloco é executado (*else*), que, nesse caso, é a Ação 3:

```
if <condição 1 verificada >:
Ação 1 executada
elif <condição 2 verificada >:
Ação 2 executada
else:
Ação 3 executada
```

Estruturas de repetição

Suponhamos agora que temos uma lista de investidores. A melhor forma de aplicar a transformação de renda em cada elemento da lista é iterado sobre ela. Suponhamos a seguinte lista:

```
investidores = [
{'nome': 'José Ramalho', 'estado': 'SP',
'renda anual': 80000.00},
{'nome': 'Jorge Lopes', 'estado': 'PA',
'renda anual': 80000.00},
{'nome': 'paulo José', 'estado': 'RJ',
'renda anual': 80000.00},
{'nome': 'Pedro Ricardo', 'estado': 'SC',
'renda anual': 80000.00},
{'nome': 'Paula Costa', 'estado': 'SC',
'renda anual': 80000.00}
]
print (investidores)
```

```
[{'nome': 'José Ramalho', 'estado': 'SP', 'renda anual': 80000.0}, {'nome': 'Jorge Lopes', 'estado': 'PA', 'renda anual': 8000
0.0}, {'nome': 'paulo José', 'estado': 'RJ', 'renda anual': 80000.0}, {'nome': 'Pedro Ricardo', 'estado': 'SC', 'renda anual':
80000.0}, {'nome': 'Paula Costa', 'estado': 'SC', 'renda anual': 80000.0}]
```

Para calcular o rendimento final dos investidores, usamos a estrutura `for` e interamos em cada elemento da lista com o trecho de código transformador:

```
transformados = []
for investidor in investidores:
 if investidor['estado'] == 'SC':
  investidor['renda final'] = \
  investidor['renda anual']*(1.0-0.15)
 elif investidor['estado'] == 'RJ':
  investidor['renda final'] = \
  investidor['renda anual']*(1.0-0.17)
transformados.append(investidor)
print (transformados)
```

```
[{'nome': 'Paula Costa', 'estado': 'SC', 'renda anual': 80000.0, 'renda final': 68000.0}]
```

Transformados, então:

```
[{'nome': 'José Ramalho', 'estado': 'SP', 'renda anual': 80000.0}, {'nome': 'Jorge Lopes', 'estado': 'PA', 'renda anual': 8000
0.0}, {'nome': 'paulo José', 'estado': 'RJ', 'renda anual': 80000.0}, {'nome': 'Pedro Ricardo', 'estado': 'SC', 'renda anual':
80000.0}, {'nome': 'Paula Costa', 'estado': 'SC', 'renda anual': 80000.0}]
```

O que fizemos aqui foi varrer a lista investidores. A cada interação associamos o elemento atual à variável investidor e aplicamos as transformações correspondentes nela.

Estrutura *for*

```
for <variável controladora> in <faixa de repetição>:
Instruções e comandos a serem executados até o fim do contador
```

Para encerrar uma linha de execução em *Python*, não é necessário nenhum tipo de caractere em especial. Deve-se apenas seguir para a linha abaixo. Entretanto, caso queira continuar um mesmo comando em uma linha abaixo, você deve utilizar o caractere barra invertida (\ ), como foi empregado no código acima, o qual foi reproduzido a seguir:

```
investidor['renda final'] = \
investidor['renda anual']*(1.0-0.15)
```

Embora estejam em linhas separadas, a barra invertida indica que fazem parte do mesmo comando.

CAPÍTULO 3

FUNÇÕES

Como utilizar

Esse código é simples, mas imagine se quisermos aplicar mais transformações. Imagine se, além de transformações no campo 'renda final', tivéssemos também transformações em outros campos com outras combinações. Por exemplo, vamos supor que diferentes estados possuem diferentes taxas, agora temos a seguinte situação:

```
taxas = {
'SC': 0.15,
'RJ': 0.17,
'SP': 0.23,
'PA': 0.11,
'RS': 0.30,
'TO': 0.17
}
print (taxas)
```

```
{'SC': 0.15, 'RJ': 0.17, 'SP': 0.23, 'PA': 0.11, 'RS': 0.3, 'TO': 0.17}
```

Lembrando da nossa lista de investidores:

```
investidores = [
{'nome': 'José Ramalho', 'estado': 'SP','renda anual': 80000.00},
{'nome': 'Jorge Lopes', 'estado': 'PA','renda anual': 80000.00},
{'nome': 'paulo José', 'estado': 'RJ','renda anual': 80000.00},
{'nome': 'Pedro Ricardo', 'estado': 'SC','renda anual': 80000.00},
{'nome': 'Paula Costa', 'estado': 'SC','renda anual': 80000.00}
]
```

Usar várias condicionais torna o código desnecessariamente extenso e muito difícil de analisar para corrigir erros ou fazer mudanças (se as regras mudarem, fica difícil atualizar as transformações). Para simplificar, usamos as funções. Uma função é um trecho de código que pode ser parametrizado e reutilizado quando necessário. Vamos criar a função que aplica taxas:

```
def aplica_taxa(investidor, taxa):
 investidor['renda final'] = investidor['renda anual']*(1.0-taxa)
 return investidor
```

A instrução `def` indica que estamos criando uma função, o nome da função é `aplica_taxa`, e, logo após, vem a lista de parâmetros. Essa lista pode ser vazia quando não há nada que a função precise para retornar um resultado (um exemplo é a função `now` do pacote `datetime`, ela não precisa de argumentos para retornar à data atual). Agora, usamos uma estrutura de repetição para implementar o cálculo da taxa usando a nova função, conforme esquema a seguir:

```
transformados = []
for investidor in investidores:
 if investidor['estado'] in taxas.keys():
  estado = investidor['estado']
  taxa = taxas[estado]
  investidor = \
  aplica_taxa(investidor, taxa)
 transformados.append(investidor)
```

Podemos visualizar os dados transformados da seguinte forma:

```
print (transformados)
```

```
[{'nome': 'José Ramalho', 'estado': 'SP', 'renda anual': 80000.0, 'renda final': 61600.0}, {'nome': 'Jorge Lopes', 'estado': 'P
A', 'renda anual': 80000.0, 'renda final': 71200.0}, {'nome': 'paulo José', 'estado': 'RJ', 'renda anual': 80000.0, 'renda fina
l': 66400.0}, {'nome': 'Pedro Ricardo', 'estado': 'SC', 'renda anual': 80000.0, 'renda final': 68000.0}, {'nome': 'Paula Cost
a', 'estado': 'SC', 'renda anual': 80000.0, 'renda final': 68000.0}]
```

Veja agora um novo cenário: se os estados do dicionário taxas possuem taxa diferenciada, mas a taxa padrão é de 30%, ainda devemos aplicar a taxa mesmo quando ela não está explicitamente no dicionário. Podemos fazer isso, portanto, incluindo a instrução `else`, conforme visto a seguir.

```
transformados = []
for investidor in investidores:
      if investidor['estado'] in taxas.keys():
       estado = investidor['estado']
       taxa = taxas[estado]
       investidor =\
       aplica_taxa(investidor, taxa)
      else:
       investidor =\
       aplica_taxa(investidor, 0.3)
      transformados.append(investidor)
print (transformados)
```

```
[{'nome': 'José Ramalho', 'estado': 'SP', 'renda anual': 80000.0, 'renda final': 61600.0}, {'nome': 'Jorge Lopes', 'estado': 'P
A', 'renda anual': 80000.0, 'renda final': 71200.0}, {'nome': 'paulo José', 'estado': 'RJ', 'renda anual': 80000.0, 'renda fina
l': 66400.0}, {'nome': 'Pedro Ricardo', 'estado': 'SC', 'renda anual': 80000.0, 'renda final': 68000.0}, {'nome': 'Paula Cost
a', 'estado': 'SC', 'renda anual': 80000.0, 'renda final': 68000.0}]
```

Exemplo de identação errada para a função `append` da tupla transformados.

```
transformados = []
for investidor in investidores:
  if investidor['estado'] in taxas.keys():
   estado = investidor['estado']
   taxa = taxas[estado]
   investidor =\
   aplica_taxa(investidor, taxa)
  else:
   investidor =\
   aplica_taxa(investidor, 0.3)
transformados.append(investidor)
print (transformados)
```

```
[{'nome': 'Paula Costa', 'estado': 'SC', 'renda anual': 80000.0, 'renda final': 68000.0}]
```

Podemos entender que o valor de taxa padrão é um parâmetro associado ao transformador. Nesse caso, é possível eliminar do trecho de código usando um valor diretamente na declaração da função.

```
def aplica_taxa(investidor, taxa=0.3):
 investidor['renda final'] = investidor['renda anual']*(1.0-taxa)
 return investidor
```

Isso informa que, se o parâmetro taxa não for informado, deve ser assumido como `0.3`. Nesse caso, o trecho de código se torna mais simples.

```
transformados = []
for investidor in investidores:
 if investidor['estado'] in taxas.keys():
  estado = investidor['estado']
  taxa = taxas[estado]
  investidor =\
  aplica_taxa(investidor, taxa)
 else:
  investidor = aplica_taxa(investidor)
 transformados.append(investidor)
print(transformados)
```

```
[{'nome': 'José Ramalho', 'estado': 'SP', 'renda anual': 80000.0, 'renda final': 61600.0}, {'nome': 'Jorge Lopes', 'estado': 'P
A', 'renda anual': 80000.0, 'renda final': 71200.0}, {'nome': 'paulo José', 'estado': 'RJ', 'renda anual': 80000.0, 'renda fina
l': 66400.0}, {'nome': 'Pedro Ricardo', 'estado': 'SC', 'renda anual': 80000.0, 'renda final': 68000.0}, {'nome': 'Paula Cost
a', 'estado': 'SC', 'renda anual': 80000.0, 'renda final': 68000.0}]
```

Mais uma refatoração que simplifica o código global é entender que a taxa por estado está ligada ao cálculo de taxas, dessa forma ela pode ser intrínseca à função, que, em lugar da taxa, deve receber como parâmetro o estado correspondente, observe na figura a seguir:

```
def aplica_taxa(investidor, estado):
 taxas = {
  'SC': 0.15,
  'RJ': 0.17,
  'SP': 0.23,
  'PA': 0.11,
  'RS': 0.30,
  'TO': 0.17
 }
 if estado in taxas.keys():
  taxa = taxas[estado]
 else:
  taxa = 0.3
 investidor['renda final'] = investidor['renda anual']*(1.0-taxa)
 return investidor
```

O dicionário taxas poderia ser declarado fora da função, existem muitas discussões sobre qual o mais adequado, mas elas ficam por conta do material complementar. Perceba que agora a função `aplica_taxa` depende apenas do investidor e do estado correspondente, com isso a estrutura de repetição pode ser ainda mais simplificada.

```
for investidor in investidores:
 estado = investidor['estado']
 investidor = aplica_taxa(investidor, estado)
```

Você deve ter percebido q ue passamos o estado como parâmetro, mas o estado já é parte do objeto investidor, ou seja, isso é uma redundância. Para eliminá-la, vamos ajustar a função `aplica_taxa`. Veja adiante:

```
def aplica_taxa(investidor):
 taxas = {
 'SC': 0.15,
 'RJ': 0.17,
 'SP': 0.23,
 'PA': 0.11,
 'RS': 0.30,
 'TO': 0.17
 }
 estado = investidor['estado']
 if estado in taxas.keys():
  taxa = taxas[estado]
 else:
  taxa = 0.3
 investidor['renda final'] = investidor['renda anual']*(1.0-taxa)
 return investidor
```

Isso transforma o trecho que itera a lista em algo ainda mais simples.

```
transformados = []
for investidor in investidores:
 investidor = aplica_taxa(investidor)
 transformados.append(investidor)
print(transformados)
```

```
[{'nome': 'José Ramalho', 'estado': 'SP', 'renda anual': 80000.0, 'renda final': 61600.0}, {'nome': 'Jorge Lopes', 'estado': 'P
A', 'renda anual': 80000.0, 'renda final': 71200.0}, {'nome': 'paulo José', 'estado': 'RJ', 'renda anual': 80000.0, 'renda fina
l': 66400.0}, {'nome': 'Pedro Ricardo', 'estado': 'SC', 'renda anual': 80000.0, 'renda final': 68000.0}, {'nome': 'Paula Cost
a', 'estado': 'SC', 'renda anual': 80000.0, 'renda final': 68000.0}]
```

Reduzimos muitas linhas a apenas duas, o preço disso foi transferir muito das decisões para o corpo da função. Porém, isso nos ajuda, agora temos um local central para a manutenção referente a taxas por estados. Como bônus, uma iteração tão simples pode ser feita em uma única instrução, o que chamamos de *compreensão de lista*.

```
transformados = [aplica_taxa(_) for _ in investidores]
```

Aqui, iteramos a lista e aplicamos a função `aplica_taxa` representada pela variável `anônima _`. Com isso, podemos obter a seguinte saída:

```
[{'nome': 'José Ramalho', 'estado': 'SP', 'renda anual': 80000.0, 'renda final': 61600.0}, {'nome': 'Jorge Lopes', 'estado': 'P
A', 'renda anual': 80000.0, 'renda final': 71200.0}, {'nome': 'paulo José', 'estado': 'RJ', 'renda anual': 80000.0, 'renda fina
l': 66400.0}, {'nome': 'Pedro Ricardo', 'estado': 'SC', 'renda anual': 80000.0, 'renda final': 68000.0}, {'nome': 'Paula Cost
a', 'estado': 'SC', 'renda anual': 80000.0, 'renda final': 68000.0}]
```

Fique atento sempre quando for copiar e colar códigos de forma direta. Muitas vezes, nesse processo, ocorrem problemas quanto à ordenação das linhas, podendo acontecer quebras em comandos que não deveriam ser interrompidos.

REFERÊNCIAS

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L. **Introduction to Algorithms**. 3. ed. Massachusetts, MA: MIT Press, 1.292 p. 2009.

ELKNER, J.; DOWNEY, A.; MEYERS, C. **How to think like a computer scientist**: learning with Python. Wickford, UK: Samurai Media Limited, 306 p. 2016.

GRUS, J. **Data science from scratch**: first principles with python. Sebastopol, CA: O'Reilly, 330 p. 2015.

HERSTEIN, N.; WINTER, D. J. **Matrix Theory and Linear Algebra**. USA: Macmillan Pub. Co., 508 p. 1988.

JOLLY, K. **Hands-on data visualization with Bokeh**: interactive web plotting for Python using Bokeh. Packt Publishing Ltd, 174 p. 2018.

MADHAVAN, S. **Mastering Python for Data Science**. USA: Packt Publishing, 294 p. 2015.

MÜLLER, A. C.; GUIDO, S. **Introduction to machine learning with Python**: a guide for data scientists. 1. ed. O'Reilly Media, 392 p. 2016.

SEDGEWICK, R.; WAYNE, K. **The textbook Algorithms**. 4, USA: Ed. Addison-Wesley Professional, 992 p. 2011.

TATSUOKA, Maurice M.; LOHNES, Paul R. **Multivariate analysis:** Techniques for educational and psychological research. USA: Macmillan Publishing Co. Inc, 479 p. 1988.