Python – fundamentos

guia rápido

Thiago Taglialegna Salles
Engenheiro Florestal (DSc.)
thiagotsalles@gmail.com

2019

Sumário

► PRIMEIROS PASSOS ◄

1. INSTALANDO O PYTHON

1.1. Downloads

1.1.1. Versões

O Python foi criado no início dos anos 90 e as versões 2 e 3 da linguagem estão em uso nos dias de hoje. Segundo o site oficial, a versão 2 terá suporte até 2020. As diferenças entre o Python 2 e 3 não são vastas e guias

1.1.2. Pacote oficial

Este material foi escrito com base na versão 3 do Python. O pacote oficial, chamado de distribuição padrão, pode ser baixado para instalação a partir de seu site oficial: www.python.org. Nele, você encontrará versões para Windows, Mac, Linux e outras plataformas. Além disso, estão disponíveis documentos sobre a linguagem e acesso a guias para ajudar quem deseja programar.

As distribuições padrão contêm o interpretador da linguagem Python, as bibliotecas padrão e um ambiente simples de desenvolvimento (IDLE). Para aplicações práticas, o ambiente de desenvolvimento incluído neste pacote é um tanto limitado e outras opções devem ser exploradas.

1.2. IDE

1.2.1. Distribuições

Integrated Development Environment, (Ambiente de Desenvolvimento Integrado), ou apenas IDE, é uma ferramenta que permite escrever e testar seu código de maneira mais fácil. Eles geralmente possuem recursos para completar trechos de código, realce de sintaxe, gerenciamento de recursos e ferramentas de depuração.

Um IDE popular e gratuito utilizado em ciência de dados é o **Spyder**. Seu site oficial é www.spyder-ide.org e há distribuições do Python que incluem sua instalação. A distribuição **Anaconda** é a maneira mais fácil de instalar o Spyder em qualquer dos sistemas operacionais suportados e a maneira

recomendada para evitar problemas inesperados. Ela pode ser obtida em www.anaconda.com. Esta distribuição inclui recursos extras de gerenciamento do Python em seu computador, bem como outros recursos para utilização da linguagem R.

Existem outras opções de IDE, como o Atom (www.atom.io), o Sublime Text (www.sublimetext.com) e o PyCharm (www.jetbrains.com/pycharm).

1.2.2. IDE online

Pode ser que você deseje executar, apresentar ou editar códigos em Python, mas está em um computador que não possui a linguagem instalada. Uma alternativa são os IDE online. Apesar de não possuírem todas as funcionalidades de um IDE padrão, eles estão disponíveis para acesso direto do navegador, sem necessidade de instalação.

Duas boas opções são o Repl.it (www.repl.it/languages) e o Coding Ground (www.tutorialspoint.com/codingground.htm). Além do Python, estas plataformas também possuem IDE para outras linguagens.

1.2.3. Realce de sintaxe

O realce de sintaxe é um recurso disponível nos IDE para melhorar a legibilidade e o contexto do código. Ele permite que o texto seja exibido em diferentes cores e fontes, de acordo com a categoria de termos.Veja os códigos abaixo:

```
@requires_authorization
def somefunc(param1='', param2=0):
    r'''A docstring'''
    if param1 > param2: # interesting
        print('Gre\'ater')
    return (param2 - param1 + 1 + 0b10l) or None

class SomeClass:
    pass

>>> message = '''interpreter
... prompt'''
```

```
@requires_authorization
def somefunc(param1='', param2=0):
    r'''A docstring'''
    if param1 > param2: # interesting
        print('Gre\'ater')
    return (param2 - param1 + 1 + 0b10l) or None

class SomeClass:
    pass

>>> message = '''interpreter
... prompt'''
```

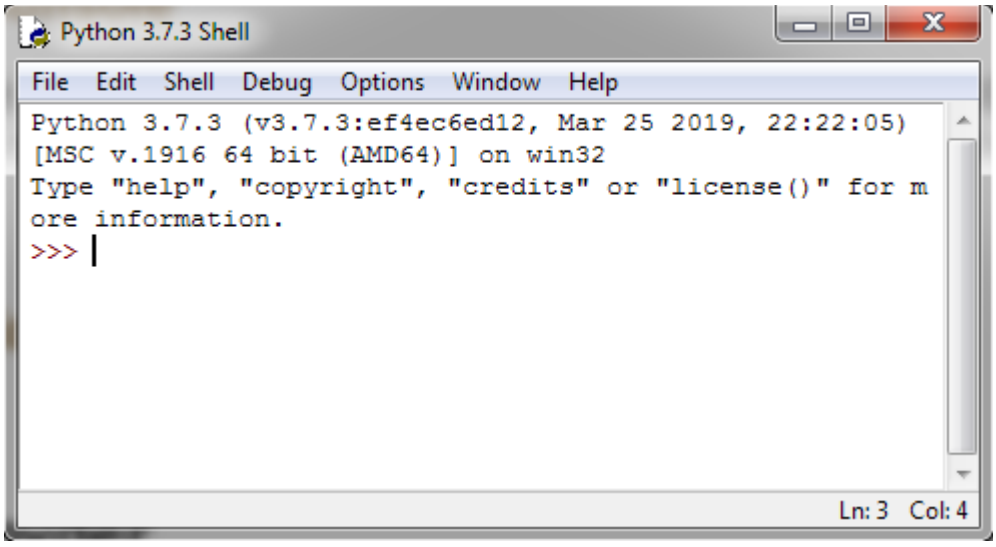
Os dois contêm o mesmo texto, mas o primeiro possui realce de sintaxe, enquanto o segundo não possui.

2. INTERFACE

2.1. Introdução aos IDE

2.1.1. IDLE

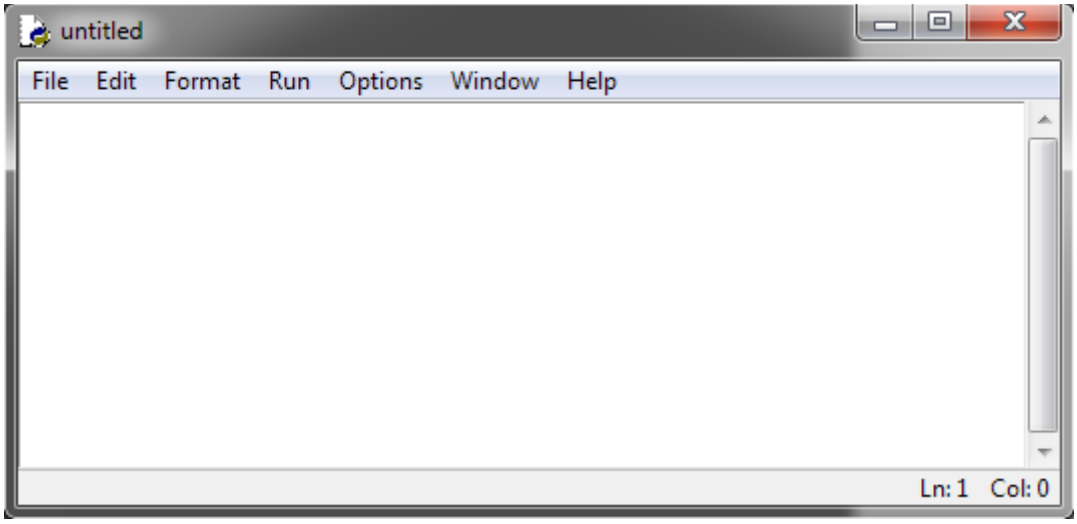
O IDLE é o ambiente de desenvolvimento integrado (IDE) padrão do Python. Ao abrir o programa, nos deparamos com o console (Shell):



Python IDLE

Nele, é possível escrever linhas de código e executá-las quando se pressiona a tecla Enter. Todavia, o console não é o lugar para se escrever e editar scripts.

No IDLE é possível criar um novo script indo em File → New File. Uma nova janela é aberta:



Novo arquivo (script) do IDLE

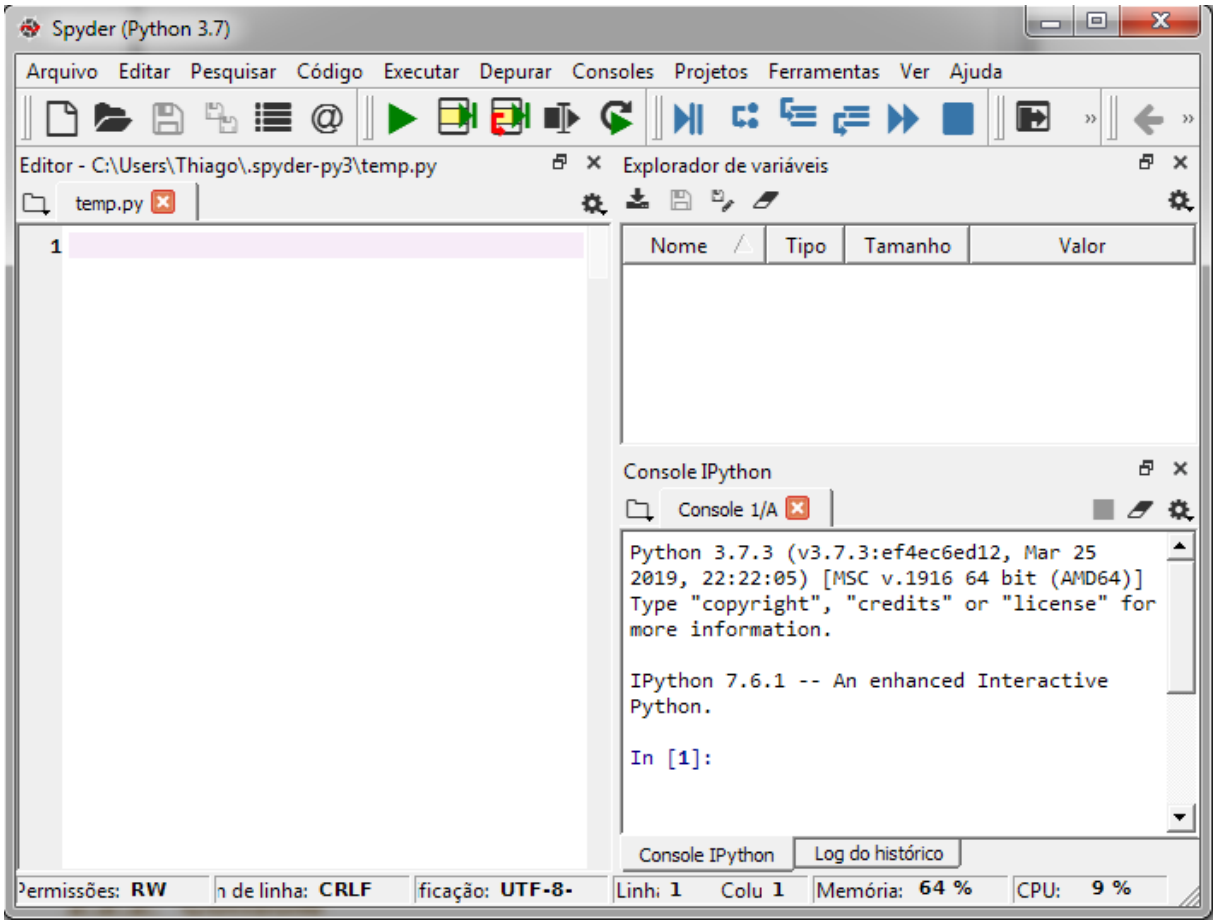
Aqui podemos escrever códigos maiores e salvar o arquivo com a extensão .py. Executamos o script todo indo em Run → Run Module, ou apertando a tecla F5. O resultado da execução aparecerá no console.

2.1.2. Spyder

O IDE do Spyder possui mais funcionalidades do que o IDE padrão (IDLE). Ao abrir o programa você vê as janelas do editor, do explorador de variáveis e do console.

A janela da esquerda é iniciada com o script em branco temp.py. Você pode abrir um script existente ou começar a escrever um novo código neste arquivo temporário. É possível executar o código todo pressionando a tecla F5 ou executar apenas um trecho de código selecionado pressionando a tecla F9.

O resultado da execução dos códigos é apresentado no console, visto na parte inferior direita da figura “IDE Spyder”. Ali também são exibidas as mensagens de erro para depuração do código e também outros resultados como gráficos produzidos na visualização de dados. Você também pode escrever e executar códigos diretamente no console.



IDE Spyder

O explorador de variáveis se encontra na janela superior direita da figura “IDE Spyder”. Ele mostra os nomes de todas as referências globais de objetos, como variáveis, funções, módulos, etc. da sessão do console atualmente selecionada. O explorador permite que você interaja com estes objetos, sendo possível editá-los sob uma interface gráfica (sem execução de código.

2.2. Sintaxe neste material

2.2.1. Código

Os trechos de código apresentados neste material estão sempre formatados com fundo escuro. O realce de sintaxe se limita a: laranja para números, amarelo para texto, cinza e itálico para comentários, roxo para verdadeiro/falso e azul para nomes de funções definidas. Veja o exemplo:

```
a = 10
b = "exemplo"
# exemplo
True
def minha_func()
```

2.2.2. Saída no console

Neste material, a representação da saída de resultados no console é feita com três sinais de maior, em vermelho, após uma linha em branco. O texto da saída não tem realce de sintaxe. Veja o exemplo:

```
print("exemplo")

>>> exemplo
```

3. BASE PARA A PRÁTICA

3.1. Iniciando

3.1.1. Significado de termos utilizados

Até aqui já mencionamos alguns termos utilizados em programação com os quais você pode não estar familiarizado. A compreensão deles e de outros termos é importante para o bom entendimento deste material. Seguem algumas definições:

Código: é um termo usado para descrever o texto que é escrito usando o protocolo específico de uma linguagem de programação.

Bloco de código: é um grupo de declarações em programação que opera como uma unidade, geralmente com seu próprio nível de escopo léxico.

Script: é uma sequência de instruções (código) que é executada por um interpretador (o Python no nosso caso). É como normalmente chamamos os arquivos com extensão .py que contêm nossos códigos.

Objeto: é todo item (variável, função, módulo, etc.) presente em uma sessão do Python. Objetos possuem atributos que lhes caracterizam e métodos que agem sobre as informações que carregam.

Função: é um grupo de instruções usado por linguagens de programação para retornar um único resultado ou um conjunto de resultados. No Python há funções padrão já implementadas, funções que podem ser importadas de módulos e funções definidas pelo programador. Elas são chamadas por um nome e seus argumentos entre parênteses, assim: func(arg1, arg2).

Iterador: é objeto que permite ao programador percorrer os itens de uma coleção de elementos, particularmente listas.

3.1.2. Métodos e atributos

Métodos são comandos aplicadas a objetos (assim como funções) e há métodos específicos para diferentes tipos de objeto. Eles são iniciados pelo caractere de ponto após o objeto sobre o qual se deseja realizar uma determinada tarefa, assim: **objeto.método()**.

Atributos são comandos que retornam características que objetos possuem. Eles também são iniciados pelo caractere de ponto após o objeto, assim: **objeto.atributo**.

3.1.3. Codificação

Pode ser que o Python em que você esteja trabalhando não aceite processar caracteres especiais como cedilhas e acentos. Isso acontece por causa da **codificação** utilizada pelo interpretador do código. A codificação chamada ASCII não permite trabalhar com caracteres especiais. Para resolver este problema, trocamos para a codificação UTF-8, iniciando nosso script com a linha:

```
# -*- coding: utf-8 -*-
```

3.2. Dúvidas e suporte

3.2.1. Ajuda

Com o Python é possível acessar a ajuda, normalmente em inglês, referente a objetos (quando disponível). Basta executar no terminal o comando help(objeto). Para funções iremos ver uma breve descrição do que fazem, os argumentos que elas tomam (com os valores padrão de argumentos opcionais) e a descrição destes argumentos. Por exemplo, para obter ajuda para a função print, fazemos:

```
help(print)

>>> Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout,
flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current
sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

Se executarmos apenas help() sem nenhum objeto entre os parênteses, entramos no modo interativo da ajuda do Python. Outra forma de conseguir ajuda em alguns IDE é clicar sobre um objeto e pressionar Ctrl + i.

3.2.2. Stack Overflow

Um local de grande ajuda quando se está com dúvidas em programação é o site www.stackoverflow.com. Ele funciona como um fórum onde a comunidade de programadores se ajuda na solução de problemas. Normalmente ao se buscar no Google alguma dúvida sobre código em Python, o Stack Overflow aparece entre os primeiros resultados.

3.2.3. Outros sites

Se você necessita estudar mais um pouco de programação, existem sites que podem lhe ajudar com isso. Por exemplo, há o DataCamp (www.datacamp.com), Udemy (www.udemy.com) e Codecademy (www.codecademy.com). Nesses sites você pode acessar cursos de Python que vão desde o básico até o avançado e muitos deles são gratuitos.

1. SINTAXE DO PYTHON

1.1. Variáveis e tipos de dados

1.1.1. Variáveis

A criação de scripts, apps web, jogos, sistemas de busca, etc. envolve armazenar e trabalhar com diferentes tipos de dados. Para isso, fazemos uso de **variáveis**. Uma **variável** é um objeto que armazena um dado e dá a ele um nome específico.

Por exemplo:

```
spam = 5
```

A variável `spam` agora armazena o número `5`.

Para que seja disponibilizada ao usuário a opção de definir o valor de uma variável durante a execução de um código, faz-se uso da função `input()`, que pode conter uma string com uma mensagem:

```
number = input("Insira um numero: ")
```

`input()` no exemplo acima pede ao usuário para inserir um número e o guarda como uma string na variável **number**. Se queremos números como resultado armazenado, devemos utilizar a função `float()` para converter a string:

```
number = float(input("Insira um numero: "))
```

1.1.2. Booleanos

Um outro tipo de dado é o chamado **booleano**. Um **booleano** é como um interruptor de luz. Ele pode ter apenas dois valores. Do mesmo modo que um interruptor pode estar apenas ligado ou desligado, um valor booleano pode ser apenas `True` (Verdadeiro) ou `False` (Falso).

Você pode usar variáveis para armazenar booleanos, assim:

```
a = True
b = False
```

1.2. Espaços em branco e indentação

1.2.1. Espaços em branco

Em Python, espaços em branco são usados para estruturar blocos de código. Espaços em branco são importantes, então você deve ter cuidado com como os usa. A estruturação de blocos com espaços é chamada de indentação (neologismo derivado da palavra em inglês indentation) ou recuo.

Você verá este erro quando sua indentação estiver incorreta:

```
IndentationError:
```

A indentação especifica a hierarquia do código quando linhas subsequentes em um bloco pertencem à sintaxe de uma linha inicial, como no exemplo abaixo:

```
def spam():
    eggs = 12
    return eggs
```

O padrão de recuo no Python é de quatro espaços, mas pode ser alterado para outro valor.

1.3. Comentários

1.3.1. Comentários de linha única

O sinal `#` é usado para incluir comentários no código. Um comentário é uma linha de texto que o Python não tentará executar como código.

Comentários tornam seu programa ou código mais fácil de entender. Quando você lê seu código, ou outros querem colaborar com você, eles podem ler seus comentários e entender o que seu código faz.

1.3.2. Comentários com Múltiplas Linhas

O sinal `#` marcará um comentário em apenas uma linha. Embora você possa escrever um comentário com mais de uma linha, iniciar cada uma delas com `#`, pode ser maçante.

Em vez disso, você pode construir comentários de múltiplas linhas, escrevendo o bloco inteiro entre um par de aspas triplas:

```
"""Olá,
eu sou um comentário de duas linhas."""
```

1.4. Números e matemática

1.4.1. Operações básicas

Você pode somar, subtrair, multiplicar, dividir números assim:

```
a = 72 + 23
b = 108 - 204
c = 108 * 0.5
d = 108 / 9
```

1.4.2. Exponenciação

Vamos trabalhar com expoentes.

```
num = 2 ** 3
```

No exemplo acima, criamos uma nova variável chamada `num` e atribuímos a ela o valor `8`, ou o resultado de 2 elevado a 3 (2^3). Note que usamos `**` em vez de `^`.

1.4.3. Operações com números inteiros

Os números podem ser armazenados como inteiros (**int**) ou no formato decimal (**float**), chamado de ponto flutuante. Para o primeiro caso, fazemos `x = 10` por exemplo. Para o segundo, `x = 10.0`. Porém, quando dividimos `x` por qualquer valor utilizando o operador **/**, o resultado será sempre do tipo float.

Para ter como resultado o valor inteiro da divisão, deve-se utilizar o operador **//**. Note que a variável `b` do exemplo abaixo irá armazenar o valor 3.

```
a = 18        # Variável armazena valor como float
b = 10 // 3   # Variável armazena valor como int
```

Um outro operador é o **resto da divisão inteira (%)**. Ele retorna o resto de uma divisão. Então, se você digitar `10 % 3`, ele retornará `1`, porque 3 cabe três vezes em 10, com resto 1.

1.5. Exibição

1.5.1. Print

A função **print** exibe no console o objeto indicado. Por exemplo, o código:

```
x = "Python"
print(x)
```

irá exibir a palavra Python no console. Uma vírgula depois de uma declaração print significa que a próxima declaração print continua exibindo o texto na mesma linha:

```
x = "python"
y = "2019"
z = "ftw"
print(x, y, z)

>>> python 2019 ftw
```

2. STRINGS E EXIBIÇÃO

2.1. Strings (texto)

2.1.1. Strings

Outro tipo de dado útil é a **string**. Uma **string** é um texto que pode conter letras, números e símbolos.

```
nome = "Rodrigo"
idade = "19"
comida = 'teste'
```

As strings devem estar entre aspas duplas ou simples.

2.1.2. Caracteres de escape

Eis alguns caracteres que causam problemas. Por exemplo:

```
'There's a snake in my boot!'
```

Este código falha porque o Python acha que a apóstrofe em `'There's'` encerra a string. Podemos usar a barra invertida para consertar o problema, assim:

```
' There\'s a snake in my boot!'
```

2.1.3. Acesso pelo índice

Cada caractere em uma string está associado a um **índice**, que por sua vez é um valor numérico. Verifique o código:

```
c = "gatos"
print(c[2])

>>> t
```

Quando executamos as duas primeiras linhas, temos como resultado a exibição do terceiro elemento da string contida na variável `c`.

No Python, contamos o índice a partir do zero em vez do um. Logo, quando fazemos um acesso utilizando o índice 2, é retornado o terceiro elemento do objeto. Além disso, o Python considera como final de um acesso por índice o número inserido - 1. Se desejamos acessar do segundo ao quarto elemento (de índice 3) da variável acima, podemos escrever `c[1:4]`.

2.2. Métodos e funções para strings

2.2.1. len()

A função `len()` retorna o número de elementos (letras) em um string. Por exemplo:

```
papagaio = "Azul Marinho"
tamanho = len(papagaio)
print(tamanho)

>>> 12
```

`len()` pode funcionar também em outros objetos que não sejam do tipo string, como listas e dicionários.

2.2.2. lower() e upper()

Você pode usar o método `lower()` para se livrar de todas as letras maiúsculas em suas strings. Você chama `lower()` assim:

```
papagaio = "Azul Marinho"
novo_papagaio = papagaio.lower()
print(novo_papagaio)

>>> azul marinho
```

O método `upper()` é a forma similar para deixar uma string totalmente em letras maiúsculas.

2.2.3. str()

Agora vamos examinar a função `str()`, que é um pouco menos direta. Ela converte outros tipos de variáveis em strings. Por exemplo:

```
str(2)
```

converte `2` em `"2"`.

2.3. Exibição avançada

2.3.1. Concatenação de strings

```
print("Vida " + "de " + "Brian")
```

Isso exibirá a frase `Vida de Brian`.

O operador **+** entre strings as somará, uma depois da outra. Note que há espaços dentro das aspas depois de `Vida` e `de` para que possamos fazer a string combinada aparecer como 3 palavras.

Combinar strings dessa forma é chamado **concatenação**.

2.3.2. Conversão de strings

Às vezes você precisa combinar uma string com algo que não é uma string. Para fazer isso, temos que converter a não-string em uma string.

```
print("Eu tenho " + str(2) + " cocos!")
```

Isso exibirá `Eu tenho 2 cocos!.`

2.3.3. Formatação de strings com %

Quando você quiser exibir uma variável em uma string, há outro modo além de concatenar. Por exemplo:

```
nome = "Mario"
print("Olá %s" % nome)

>>> Olá Mario
```

O operador **%** depois de uma string é usado para combinar uma string com variáveis. O operador **%** substituirá um **%s** na string pela variável string que vem depois dele.

Você precisa do mesmo número de %s em uma string que o número de variáveis para combinação:

```
tempo = "9"
unidade = "horas"
print("São %s %s" % (tempo, unidade))

# Isso exibirá "São 9 horas"
```

2.4. Data e hora

2.4.1. Obtendo a data e hora atuais

Podemos usar uma função chamada **datetime.now()** para conseguir a data e hora atuais.

```
from datetime import datetime  # Importação de função
print(datetime.now())
```

A primeira linha importa a função **datetime** do módulo de mesmo nome para que possamos usá-la. A importação de funções e módulos é vista em mais detalhes mais adiante neste material. A segunda linha exibe a data e hora atuais.

A saída se parece com `2018-11-25 23:45:14.317454`.

2.4.2. Extraindo informações de data

E se você não quiser a data e hora completos? Veja o exemplo:

```
from datetime import datetime
agora = datetime.now()
ano = agora.year
mes = agora.month
dia = agora.day
```

Nas duas primeiras linhas importamos o módulo datetime e guardamos a informação de hora e data atuais na variável `agora`. A variável passa a possuir atributos (year, month e day) que podem ser acessados. A partir da terceira linha, tomamos o ano, o mês e o dia armazenados em `agora` por meio destes atributos.

2.4.3. Exibição de data formatada

E se quisermos exibir a data de hoje no seguinte formato? `mm/dd/yyyy`. Vamos usar a substituição de strings de novo.

```
from datetime import datetime
agora = datetime.now()

print('%s-%s-%s' % (agora.day, agora.month, agora.year))
# Vai exibir 19-02-2019 por exemplo
```

Lembre-se que o operador `%` preencherá os espaços temporários `%s` na string à esquerda com as strings nos parênteses à direita. Seguindo a mesma ideia, o exemplo abaixo exibirá a data, e então, em uma linha separada, a hora.

```
from datetime import datetime
agora = datetime.now()

print('%s/%s/%s' % (agora.day, agora.month, agora.year))
print('%s:%s:%s' % (agora.hour, agora.minute, agora.second))
```

3. FLUXO DE CONTROLE E CONDIÇÕES

3.1. Comparadores

3.1.1. Tipos de comparadores

Vamos começar com o aspecto mais simples do controle de fluxo: **comparadores**. Existem seis: igual a: **==**, diferente de: **!=** , menor do que: **<**, menor ou igual a: **<=**, maior que: **>**, maior ou igual a: **>=**.

Comparadores verificam se um valor é (ou não) igual, maior do que (ou igual a), ou menor que (ou igual a) outro valor. Fica definido então que "==" compara se duas coisas são iguais, e "=" atribui um valor a uma variável.

3.1.2. Resultados dos comparadores

Comparações resultam em True (Verdadeiro) ou False (Falso), que são valores booleanos. As expressões abaixo resultarão em `True`, `False`, `True` e `False`.

```
bool_um = 3 < 5
bool_dois = 3 == 5
bool_tres = 3 != 5
bool_quatro = 3 > 5
```

3.2. Operadores booleanos

3.2.1. Tipos de operadores booleanos

Operadores booleanos comparam declarações e resultam em valores booleanos. Há três operadores booleanos:

`and` (e), que verifica se as duas afirmações são `True`;

`or` (ou), que verifica se pelo menos uma das afirmações é `True`; e

`not` (não), que inverte o booleano de uma operação.

3.2.2. And

O operador booleano `and` (e) retorna `True` (verdadeiro) quando as expressões dos dois lados do `and` são verdadeiros. Por exemplo:

`1 < 2 and 2 < 3` é `True`;

`1 < 2 and 2 > 3` é `False`.

3.2.3. Or

O operador booleano `or` (ou) retorna `True` (verdadeiro) quando pelo uma das expressões ligadas por `or` for verdadeira. Por exemplo:

`1 < 2 or 2 > 3` é `True`;

`1 > 2 or 2 > 3` é `False`.

3.2.4. Not

O operador booleano `not` retorna `True` para declarações falsas e `False` declarações verdadeiras. Por exemplo:

`not False` resultará em `True`, enquanto `not 41 > 40` retornará `False`.

3.2.5. Ordem dos operadores

Operadores booleanos não são simplesmente avaliados da esquerda para a direita. Assim como os operadores aritméticos, há uma ordem na execução dos operadores booleanos:

`not` é o primeiro a ser avaliado; `and` é o segundo a ser avaliado; e `or` é o último a ser avaliado.

Por exemplo, `True or not False and False` retorna `True`.

Parênteses `()` garantem que suas expressões sejam avaliadas na ordem que você quer. Qualquer coisa entre parênteses é avaliada como uma unidade separada.

3.3. If, else e elif

3.3.1. If e sintaxe das declarações condicionais

`if` é uma declaração condicional que executa um código especificado depois de verificar se sua expressão é `True` (verdadeira). Eis um exemplo da sintaxe da declaração `if`:

```
if 8 < 9:
    print("Oito é menor que 9!")
```

Neste exemplo, `8 < 9` e é a expressão verificada e `print("Oito é menor que nove!")` é o código especificado. Caso `8 < 9` retorne `True`, então o bloco recuado de código depois da expressão será executado. Caso retorne `False`, o bloco recuado será pulado.

3.3.2. Else

A declaração `else` complementa a declaração `if`. Um par if/else diz: "Se esta expressão for verdadeira, execute o bloco de código recuado; caso contrário, execute o código depois da declaração else."

Ao contrário de if, else não depende de uma expressão. Por exemplo:

```
if x > y:
    print("x é maior que y!")
else:
    print("x não é maior que y!")
```

3.3.3. Elif

`elif` é uma abreviação para "else if." Isso significa exatamente o que parece: "caso contrário, se a expressão a seguir for verdadeira, faça isto!"

```
if x > y:
    print("x é maior que y")
elif x < y:
    print("x é menor que y")
else:
    print("x é igual a y")
```

No exemplo acima, a declaração `elif` é verificada apenas se a declaração `if` original for `False`.

4. FUNÇÕES

4.1. Sintaxe das funções

4.1.1. Partes das Funções

Funções são definidas basicamente com duas partes. O **header** (cabeçalho), que inclui a palavra-chave **def**, o nome da função e quaisquer **parâmetros** de que a função precise. Por exemplo:

```
def mensagem():   # Não há parâmetros
```

E o **corpo**, que descreve os procedimentos que a função executa. O corpo é feito em **linhas recuadas**, assim como as declarações condicionais. Por exemplo:

```
print("Oi mundo")
```

Eis a função completa:

```
def mensagem():   # Não há parâmetros
    print("Oi mundo")

mensagem()  # Vai exibir Oi mundo
```

Depois que uma função é definida, ela pode ser **chamada** para ser implementada. No código anterior, `mensagem()` na última linha diz ao programa para procurar pela função chamada `mensagem` e executar o código dentro dela.

Podemos utilizar a função `return` no lugar de `print` para que um valor seja retornado pela função ao invés de exibido no console. Isso permite que o resultado de uma função seja associado a uma variável por exemplo.

4.1.2. Parâmetros e argumentos

Vamos examinar uma linha que define uma função `quadrado`:

```
def quadrado(n):
```

`n` é um **parâmetro** de `quadrado`. Um parâmetro age como um nome de variável para um **argumento** inserido. No exemplo abaixo, chamamos `quadrado` com o argumento `10`.

```
def quadrado(n):
    # Retorna o quadrado de um número
    result = n ** 2
    print("%s ao quadrado é %s" % (n, result))

quadrado(10)

>>> 10 ao quadrado é 100
```

Uma função pode exigir quantos parâmetros você quiser, mas quando você chama a função, geralmente deve inserir um número correspondente de argumentos.

4.1.3. Funções Chamando Funções

Uma função pode chamar outra função:

```
def fun_um(x):
    return x * 5

def fun_dois(y):
    return fun_um(y) + 7
```

É completamente válido chamar a função `fun_um(x)` com a variável `y`. Passamos o valor de `y` para a nova função no argumento `x`. Se fizermos `a = fun_dois(2)`, a variável `a` irá receber o valor `17`.

4.2. Importação de módulos

Um módulo é um arquivo que contém definições - incluindo variáveis e funções - que você pode usar depois de importadas.

4.2.1. Importações genéricas

Há um módulo do Python chamado `math` que inclui diversas variáveis e funções úteis, e `sqrt()` (raiz quadrada) é uma dessas funções. Para acessar `math`, tudo o que você precisar é a palavra-chave `import`. Quando você importa um módulo desta forma, isto é chamado de **importação genérica**. Abaixo segue um exemplo de uso do módulo math:

```
import math
print(math.sqrt(25))  # Exibe a raiz quadrada de 25
```

Outro módulo é o `random`, do qual pode ser importada a função `randint(x, y)`, onde `x` e `y` são os limites para se gerar um número inteiro aleatório.

```
import random
a = random.randint(1, 10)
print(a)  # Exibe um número aleatório entre 1 e 10 salvo em a
```

4.2.2. Importações de função

Quando precisamos somente da função `sqrt`, por exemplo, pode ser frustrante ficar digitando `math.sqrt()` o tempo todo.

É possível importar apenas certas variáveis ou funções de um dado módulo. Obter apenas uma única função de um módulo é chamado **importação de função**, e é feito com a palavra-chave `from`:

```
from module import function
```

Assim, você pode simplesmente digitar `sqrt()` para obter a raiz quadrada de um número, como abaixo:

```
from math import sqrt
print(sqrt(25))  # Exibe a raiz quadrada de 25
```

4.2.3. Importações universais

E se ainda quisermos todas as funções e variáveis de um módulo, mas não quisermos digitar constantemente `math.`? A **importação universal** pode tratar disso. A sintaxe é:

```
from module import *
```

Importações universais podem parecer ótimas em teoria, mas não são uma boa ideia por um motivo muito importante: elas enchem seu programa com uma quantidade grande de nomes de funções e variáveis sem a segurança desses nomes ainda estarem associados com o(s) módulo(s) de onde eles vieram.

Se você tiver uma função própria chamada `sqrt` e importar `math`, sua função está segura: há o seu `sqrt` e há `math.sqrt`. Entretanto, se você usar `from math import *`, você terá um problema: duas funções diferentes com exatamente o mesmo nome.

Mesmo se suas próprias definições não entrarem em conflito direto com os nomes de módulos importados, se você usar `import *` de diversos

módulos ao mesmo tempo, não será capaz de descobrir que variável ou função veio de onde.

Por essas razões, é melhor usar ou `import module` e digitar `module.name` ou simplesmente usar `import` para variáveis e funções específicas de diversos módulos conforme necessário.

O código abaixo mostrará a tudo o que está disponível no módulo `math`.

```
import math
everything = dir(math)
print everything
```

4.2.4. Renomeando módulos importados

Uma maneira de aliviar o trabalho de repetição ao escrever funções de módulos importados de forma genérica é dando nomes mais simples a estes módulos. Para isso, usamos a expressão `as`.

Para exibir um número aleatório entre 0 e 5 podemos então escrever:

```
import random as rd
rd.randint(0, 5)
```

4.3. Funções embutidas

Existem algumas funções embutidas no Python (não são necessários módulos), como por exemplo `input()`. Há as que usamos com strings, como `upper()`,`lower()`, `str()`, e `len()` e também existem outras de cunho analítico.

4.3.1. max(), min() e abs()

A função `max()` toma qualquer número de argumentos e retorna o maior deles. Como "maior" aqui pode ter definições estranhas, é melhor usar `max()` em números inteiros e de ponto flutuante, onde os resultados são simples, não em outros objetos, como strings.

Por exemplo, `max(1,2,3)` retornará `3` (o maior número no conjunto de argumentos). `min()` retorna então o menor de uma dada série de argumentos. Já a função `abs()` retorna o **módulo** do número que toma como argumento. Ao contrário de `max()` e `min()`, `abs()` toma apenas um único número.

4.3.2. type()

A função `type()` retorna o **tipo** dos dados que recebe como um argumento. Se você pedir ao Python para fazer o seguinte:

```
print(type(42))
print(type(4.2))
print(type('spam'))
```

O resultado do Python será:

```
>>> <class 'int'>
>>> <class 'float'>
>>> <class 'str'>
```

5. LISTAS E DICIONÁRIOS

5.1. Listas

5.1.1. Introdução às listas

Listas são um tipo de dado que você pode usar para armazenar uma coleção de diferentes informações como uma sequência sob um único nome de variável. Você pode atribuir itens a uma lista com uma expressão na forma:

```
minha_lista = [item_1, item_2]
```

com os itens entre colchetes. Uma lista também pode estar vazia: `lista_vazia = []`. Uma lista também pode conter outras listas:

```
minha_lista = [[item_1, item_2],[item_2, item_3],[item_4, item_5]]
```

Listas são muito similares a strings, mas há algumas diferenças essenciais.

5.1.2. Acesso pelos índices

Você pode acessar um item individual na lista pelo seu **índice**. Um índice é como um endereço que identifica a posição do item na lista. Usa-se o diretamente depois do nome da lista, entre colchetes, assim: `minha_lista[índice]`.

Lembre-se que os índices começam em 0, não 1. Você acessa o primeiro item de uma lista dessa forma: `minha_lista[0]`. O segundo item em uma lista está no índice 1: `minha_lista[1]`.

Um índice de lista se comporta como qualquer outro nome de variável. Ele pode ser usado para acessar e atribuir valores.

```
animais = ["pinguim", "coelho", "cavalo"]
print(animais[0])     # Exibe o valor "pinguim"

animais[2] = "hiena"  # Muda "cavalo" para "hiena"
print(animais[2])     # Exibe o valor "hiena"
```

Os índices podem ser utilizados para acessar itens em mais de um nível em uma lista.

```
minha_lista = [[item_1, item_2],[item_2, item_3],[item_4, item_5]]
print(minha_lista[2][0])
```

O código acima exibe `item_4`, que é o primeiro elemento do terceiro item de `minha_lista`.

5.2. Capacidades e funções de lista

5.2.1. Chegadas tardias e comprimento da lista

Uma lista não precisa ter um comprimento fixo. Você pode adicionar itens ao fim de uma lista sempre que quiser.

```
letras = ['a', 'b', 'c']
letras.append('d')
print(len(letras))
print(letras)
```

No exemplo acima, primeiro criamos uma lista chamada `letras`. Então, adicionamos a string `'d'` ao fim da lista `letras` através do método `append()`. Depois, exibimos `4`, o comprimento da lista `letras`. Finalmente, exibimos `['a', 'b', 'c', 'd']`.

5.2.2. Buscar índice de item em lista

Às vezes você precisar procurar por um item em uma lista.

```
animais = ["cachorro", "morcego", "gato"]
print(animais.index("morcego"))
```

Primeiro, criamos uma lista chamada `animas` com três strings. Então, através do método `index()`, exibimos o primeiro índice que contém a string `"morcego"`, o que exibirá `1`.

5.2.3. Inserir item num determinado índice da lista

Também podemos inserir itens em uma lista. Considerando a lista anterior:

```
animais.insert(1, "coelho")
print(animais)
```

Inserimos `"coelho"` no índice 1 através do método `insert()`. Desta forma, os outros elementos foram movidos na lista. Exibimos `["cachorro", "coelho", "morcego", "gato"]`.

5.2.4. Removendo itens

Às vezes você precisa remover algo de uma lista.

```
beatles = ["john", "paul", "george", "ringo", "stuart"]
beatles.remove("stuart")
print(beatles)

>>> ["john", "paul", "george", "ringo"]
```

Criamos uma lista chamada `beatles` com 5 strings. Então, removemos o item de `beatles` que corresponda à string `"stuart"`. Note que o método `remove(item)` não retorna nada. Finalmente, exibimos a lista para confirmar que `"stuart"` foi realmente removido.

5.2.5. Organizando uma lista

Se sua lista estiver uma bagunça, você pode precisar organizá-la (`sort()`).

```
animais = ["gato", "cachorro", "morcego"]
animais.sort()

print(animais)
```

No código acima, criamos uma lista chamada `animais` com três strings. As strings não estão em ordem alfabética. Então, colocamos `animais` em ordem alfabética. Note que `sort()` modifica a lista em vez de retornar uma nova lista. Então, exibimos a lista `['cachorro', 'gato', 'morcego']`.

5.2.6. Unindo os elementos

O método `join()` une os elementos de uma lista com um argumento definido entre elas.

```
letras = ['a', 'b', 'c', 'd']
print(" ".join(letras))
print("---".join(letras))
```

No exemplo acima, criamos uma lista chamada `letras`. Então, exibimos `a b c d`. O método `join()` usa a string para combinar os itens na lista. Finalmente, exibimos `a---b---c---d`.

Lembre-se que a lista não se torna uma string. Para tal, deve-se criar uma nova variável e igualá-la ao resultado de `join()`.

5.3. Fatiamento de lista

5.3.1. Sintaxe

Às vezes, queremos apenas parte de uma lista no Python. Talvez queiramos apenas os primeiros elementos; talvez queiramos apenas os últimos. Talvez queiramos elementos alternados. O fatiamento de lista nos permite acessar os elementos de uma lista de modo conciso. A sintaxe é assim:

```
[start:end:stride]
```

Onde `start` descreve o ponto de início do fatiamento (inclusive), `end` é onde ele termina (exclusive), e `stride` descreve o espaço entre itens na lista fatiada. Por exemplo, um passo (stride) de 2 selecionaria itens alternados de uma lista.

Esta funcionalidade de fatiamento também é aplicável a strings e outros objetos.

5.3.2. Acessando somente parte de uma lista

Considere o código:

```
letras = ['a', 'b', 'c', 'd', 'e']
fatia = letras[1:3]
print(fatia)
print(letras)
```

No exemplo acima, criamos primeiro uma lista chamada `letras`. Então, tomamos uma subseção e a armazenamos na lista `fatia`. Começamos no índice antes dos dois pontos e continuamos até, **mas não incluindo**, o índice depois dos dois pontos. A seguir, exibimos `['b', 'c']`. Lembre-se que começamos a contar índices a partir do 0 e paramos **antes** do índice 3. Finalmente, exibimos `['a','b', 'c', 'd', 'e']`, apenas para mostrar que não modificamos a lista original `letras`.

5.3.3. Omitindo os índices

Se sua fatia incluir a partir do primeiro ou até o último item de uma lista (ou outro objeto), o índice não precisa ser incluído.

```
letras[:2]
# Toma os dois primeiros itens
letras[3:]
# Toma os itens a partir do quarto elemento
letras[::2]
# Toma todos os itens considerando um salto de 2
```

Ao se omitir um índice, é considerado o valor padrão. O índice padrão de início é `0`, índice de término padrão é o fim da lista e o passo padrão é `1`.

5.3.4. Invertendo uma lista

Um passo **negativo** percorre a lista da direita para a esquerda.

```
letras = ['A', 'B', 'C', 'D', 'E']
print(letras[::-1])
```

No exemplo acima, exibimos `['E', 'D', 'C', 'B', 'A']`.

Com a mesma lógica, se executarmos `print(letras[-2:])`, iremos exibir `['D', 'E']`.

5.4. Dicionários

5.4.1. Introdução aos dicionários

Um dicionário é similar a uma lista, mas você acessa valores procurando uma **chave** em vez de um índice. Uma chave pode ser qualquer string ou número (apenas um elemento). Dicionários são contidos entre chaves, assim:

```
dic = {'ch_1' : 1, 'ch_2': 2, 'ch_3': 3}
```

Este é um dicionário chamado `dic` com três **pares chave-valor**. A chave `'ch_1'` aponta para o valor `1`, `'ch_2'` para `2`, e assim por diante.

O comprimento `len()` de um dicionário é o número de pares chave-valor que ele tem. Cada par é contado apenas uma vez, mesmo se o valor for uma lista (Isso mesmo: você pode colocar listas **dentro** de dicionários).

Dicionários são úteis para coisas como listas de telefone (combinando um nome a um número de telefone) ou páginas de acesso de websites (combinando um endereço de e-mail com um nome de usuário).

5.4.2. Novas entradas

Como as listas, os dicionários são mutáveis. Isso significa que eles podem ser modificados depois de serem criados. Uma vantagem disso é que podemos adicionar novos pares chave/valor ao dicionário depois que ele é criado, assim:

```
dic[nova_chave] = novo_valor
```

Exemplo:

```
menu = {}  # Dicionário vazio
menu['Frango'] = 14.50  # Adicionando um novo par chave-valor
```

Um par vazio de chaves `{}` é um dicionário vazio, assim como um par vazio de `[]` é uma lista vazia.

5.4.3. Deletando chaves e trocando valores

Como os dicionários são mutáveis, eles podem ser mudados de muitas formas. Itens podem ser removidos de um dicionário com o comando `del`:

```
del dic[chave]
```

removerá a chave `chave` e o valor associado a ela.

Um novo valor pode ser associado a uma chave atribuindo-se um valor a ela. Veja o exemplo:

```
dic = {'ch_1' : 1, 'ch_2': 2, 'ch_3': 3}
dic['ch_2'] = 7
print(dic['ch_2'])

>>> 7
```

5.4.4. Removendo itens em listas dentro de dicionários

Às vezes você precisa remover algo de uma lista dentro de um dicionário. Veja o exemplo de como fazer isso:

```
meu_dic = {
    'dinheiro': 500,
    'mochila': ['canivete', 'saco de dormir', 'pão']
}  # Um dicionário com 2 chaves, escrito de maneira fácil de se ler

meu_dic['mochila'].remove('canivete')
# Remove o item 'canivete' contido na lista da chave 'mochila'
```

5.4.5. Método items()

Lembrando que um dicionário é uma coleção de chaves e valores, o iterador `.items()` funciona da seguinte forma:

```
dic = {
    "Nome": "Guido",
    "ID": 56,
    "BDFL": True
}

print(dic.items())
# Exibe dict_items([('Nome', 'Guido'), ('ID', 56), ('BDFL', True)])
```

`items()` retorna um iterador (`dict_items`) na forma de uma lista de **tuplas** com cada tupla contendo de um par chave/valor do dicionário. Uma tupla pode ser pensada como uma lista imutável (embora isso seja uma simplificação); tuplas são cercadas por parênteses e podem conter qualquer tipo de dado.

5.4.6. keys() e values()

O método `keys()` retorna um iterador (`dict_keys`) na forma de uma lista com as chaves do dicionário, e o método `values()` retorna um iterador (`dict_values`) na forma de uma lista com os valores do dicionário.

Por exemplo, podemos aplicar estes métodos ao dicionário do item anterior:

```
print(dic.keys())
print(dict.values())

>>> dict_keys(['Nome', 'IDADE', 'BDFL'])
>>> dict_values(['Guido', 56, True])
```

Para armazenarmos o conteúdo dos iteradores em variáveis, podemos utilizar a função `list()`. Ainda considerando `dic` do item anterior:

```
chaves = list(dic.keys())
# Armazena a lista ['Nome', 'IDADE', 'BDFL']
valores = list(dic.values())
# Armazena a lista ['Guido', 56, True]
```

6. LAÇOS

6.1. Laços for

6.1.1. Fazendo algo com cada item de um objeto

Se você quiser fazer algo com todos os itens em uma lista, string, etc., pode usar um laço `for` seguindo a sintaxe:

```
for item in objeto:
```

Um nome de variável qualquer (`item`) se segue à palavra-chave `for`; esta variável receberá um valor por vez do objeto do laço. Logo, `in objeto` designa que o laço vai atuar sobre `objeto`. A linha termina com dois pontos (`:`) e segue-se com um código recuado que será executado uma vez por item de `objeto`.

Podemos usar o comando `break` para sair de um laço for. Mais sobre esse comando é mostrado na função `while`. O comando `else` também pode ser

utilizado ao final de `for` para executar um código ao final da execução do laço, mas apenas se o `for` terminar normalmente - ou seja, não com um `break`. A declaração `else` deve estar com o mesmo recuo de `for`.

6.1.2. Loop com laços

Os laços podem ser utilizados para dar loop em um conjunto de código. Por exemplo, o código abaixo cria uma lista e depois percorre a mesma exibindo no console cada um de seus itens:

```
minha_lista = [12, 5, 8, 0]

for x in minha_lista:
    print(x)

>>> 12
5
8
0
```

Também podemos utilizar um iterador em um loop para repetir uma função, por exemplo:

```
for x in range(4):
    print("loop")
```

O código acima irá percorrer um iterador de 0 a 3 e irá exibir a palavra loop 4 vezes.

6.1.3. Percorrendo um dicionário

Como funciona percorrer um dicionário com um laço? Você deveria obter a chave ou o valor? A resposta curta é: você obtém a chave, que pode usar para obter o valor. Veja o código abaixo:

```
dic = {'x': 9, 'y': 10, 'z': 20}
for key in dic:
    if dic[key] == 10
        print("Este dicionário tem o valor 10!")
```

Primeiro criamos um dicionário com strings como chaves e números como valores. Então percorremos o dicionário, armazenando a cada vez a chave em `key`. A cada armazenamento, verificamos se o valor associado à chave é igual a 10. Quando verdadeiro, exibimos `Este dicionário tem o valor 10!`

6.1.4. Função enumerate()

Uma desvantagem de usar o laço **for** é que você não sabe o índice do que está examinando. Geralmente, isso não é um problema, mas às vezes é útil saber em que ponto da lista você está. A função embutida `enumerate` nos ajuda com isso.

`enumerate` funciona fornecendo um índice correspondente a cada elemento na lista (ou string, etc.) que você está percorrendo. Observando o código abaixo:

```
menu = ['pizza', 'massa', 'salada', 'nachos']

for indice, item in enumerate(menu, 1):
    print(indice, item)
```

Sempre que você passar pelo laço, `indice` será incrementado em um (iniciado pelo argumento 1 da função enumerate, que pode ser deixado em branco para a contagem iniciar em 0), e `item` será o próximo item na sequência. É muito similar a usar um laço `for` normal com uma lista, exceto que isso permite um modo fácil de contar quantos itens foram vistos. O código acima resultará em:

```
>>> 1 pizza
2 massa
3 salada
4 nachos
```

6.1.5. Laço em duas listas ao mesmo tempo

Aqui a função embutida `zip` se torna útil. `zip` criará pares de elementos quando são usadas duas listas, parando no fim da lista mais curta. `zip` também pode tratar três ou mais listas. O código abaixo compara os pares de mesmo índice nas listas e exibe o de maior valor:

```
lista_a = [3, 9, 17, 15, 19]
lista_b = [2, 4, 8, 10, 30, 40, 50, 60, 70, 80, 90]

for a, b in zip(lista_a, lista_b):
    print(max(a, b))

>>> 3
9
17
15
30
```

6.2. Laços while

6.2.1. Funcionamento de while

O laço **while** é similar a uma declaração **if**: ela executa o código no seu interior se certa condição for verdadeira. A diferença é que o laço while continuará a ser executado enquanto a condição for verdadeira. Em outras palavras, em vez de ser executado se (if) uma condição for verdadeira, o código será executado enquanto (while) a condição for verdadeira.

No código abaixo, enquanto `count` for menor do que 20, o laço (bloco de código recuado) continuará a ser executado. A linha 3 aumenta `count` em 1.

```
while count < 10:
    print("Olá, sou um while e a contagem é", count)
    count += 1
```

Da mesma forma que em **for**, a declaração **else** pode ser utilizada em while, executando o comando ao final do laço, contanto que este não seja interrompido por um break.

6.2.2. Condição em while

A **condição** é a expressão que decide se o laço será executado ou não. Há 5 etapas no código abaixo:

```
cond = True

while cond:
    print("Sou um loop")
    cond = False

>>> Sou um loop
```

1 - A variável `cond` é igualada a `True`. 2 - O laço `while` verifica se `cond` é `True` (verdadeira). Ela é, então o programa entra no laço. 3 - A declaração `print` é executada. 4 - A variável `cond` é igualada a `False`. 5 - O laço `while` verifica novamente se `cond` é `True`. Ela não é, por isso o laço não é executado novamente.

6.2.3. Break

Break é uma declaração de uma linha que significa "saia do laço atual". O código abaixo possui um laço `while` com uma condição que é sempre verdade. Como resultado, ele exibe os números de 0 a 9. Caso não houvesse o `break` ao final, ocorreria um loop infinito.

```
cont = 0

while True:
    print(cont)
    cont += 1
    if cont >= 10:
        break
```

7. TÓPICOS AVANÇADOS EM LISTAS E DICIONÁRIOS

7.1. Compreensão de listas e dicionários

7.1.1. Sintaxe para listas

A compreensão de lista permite a criação deste tipo de objeto a partir de um laço **for**. Eis um exemplo simples da sintaxe:

```
minha_lista = [x for x in range(1, 6)]
print(minha_lista)  # Exibe [1, 2, 3, 4, 5]
```

Isso criará uma lista (`minha_lista`) com os números inteiros de um a cinco. Se você quiser o dobro desses números, pode usar:

```
minha_lista = [1, 2, 3, 4, 5]
nova_lista = [x * 2 for x in minha_lista]
print(nova_lista)  # Exibe [2, 4, 6, 8, 10]
```

7.1.2. Compreensão com mais de uma lista

Pode ser que você queira utilizar mais de um objeto na compreensão de listas. Por exemplo, você pode desejar salvar em uma variável o resultado da multiplicação entre os elementos de duas listas. Veja o código:

```
lista_1 = [1, 2, 3]
lista_2 = [3, 3, 3]
lista_3 = [x * y for x, y in zip(lista_1, lista_2)]
print(lista_3)

>>> [3, 6, 9]
```

Repare que para conseguir realizar a construção de `lista_3` utilizamos a função `zip()`. Ela une objetos iteráveis e é útil em compreensão de listas e dicionários.

7.1.3. Sintaxe para dicionários

A mesma lógica das listas pode ser utilizada para montar dicionários. Como temos dois elementos (chave e valor) nos itens de um dicionário, também teremos dois elementos na compreensão:

```
chaves = [1, 2, 3, 4]
valores = ["primeiro", "segundo", "terceiro", "quarto"]

dic = {k: v for k, v in zip(chaves, valores)}
print(dic)

>>> {1: 'primeiro', 2: 'segundo', 3: 'terceiro', 4: 'quarto'}
```

Aqui, como esperado, temos a chave e o valor separados por dois pontos na construção do dicionário. Em situações mais complexas, você pode até utilizar uma expressão de compreensão de lista para gerar os valores de seu dicionário.

7.2. Condicionais em compreensão

7.2.1. If em listas e dicionários

Pode ser que você deseje adicionar elementos em sua compreensão de lista apenas se uma condição for verdadeira. Por exemplo, dos números de 1 a 20, pegamos apenas aqueles que são divisíveis por três:

```
lista_por_3 = [x for x in range(1, 20) if x % 3 == 0]
print(lista_por_3)

>>> [3, 6, 9, 12, 15, 18]
```

Na primeira parte da compreensão de lista acima, pedimos para adicionar o valor (`x`) de um iterador com números inteiros de 1 a 20. Na segunda parte (`if`) determinamos que a primeira parte só será executada se o resto da divisão de x por 3 for igual a zero (ou seja, `x` é divisível por 3).

7.2.2. If/else em listas e dicionários

Também é possível utilizar a compreensão adicionando certos valores quando uma condição é verdadeira e outros valores quando a condição é falsa. Por exemplo:

```
chaves = list(range(1, 8))
valores = [3, 12, 7, 44, 1, 4, 2, 25]
dicio = {k: v if v >= 5 else "< 5" for k, v in zip(chaves, valores)}
```

Aqui as declarações de if e else são colocadas antes do laço for. Caso se se deseje aplicar condições à chave do dicionário, as declarações deverão ser feitas antes dos dois pontos. Se imprimirmos dicio, será exibido `{1: '< 5', 2: 12, 3: 7, 4: 44, 5: '< 5', 6: '< 5', 7: '< 5'}`.

7.3. Introdução ao uso de funções com listas

7.3.1. Passando uma lista para uma função

É possível passar uma função do mesmo modo que se passa qualquer outro argumento para uma função. Por exemplo:

```
def list_func(x):
    return x

n = [3, 5, 7]
print(list_func(n))
```

O código acima resultará em `[3, 5, 7]`.

7.3.2. Usando um elemento de uma lista em uma função

Passar uma lista para uma função a armazenará no argumento (do mesmo modo que uma string ou um número).

```
def primeiro(items):
    print(items[0])

num = [2, 7, 9]
primeiro(num)
```

No exemplo acima, definimos uma função chamada `primeiro`. Ela tem um argumento chamado `items`. Dentro da função, exibimos (`print`) o item armazenado no índice zero de `items`. Depois da função, criamos uma nova lista chamada `num`. Finalmente, chamamos a função `primeiro` com `num` como argumento, que exibirá `2`.

7.3.3. Modificando um elemento de uma lista em uma função

Modificar um elemento em uma lista dentro de uma função é o mesmo que se você estivesse apenas modificando um elemento de uma lista fora de uma função.

```
def dobro_prim(n):
    n[0] = n[0] * 2

num = [1, 2, 3, 4]
dobro_prim(num)
print(numbers)
```

Criamos uma lista chamada `num`. Usamos a função `dobro_prim` para modificar essa lista. finalmente exibimos `[2, 2, 3, 4]`.

7.4. Usando a lista inteira em uma função

7.4.1. A melhor forma de percorrer uma lista em uma função

Temos dois modos de percorrer uma lista:

```
for item in lista:
    print(item)
```

e

```
for i in range(len(lista)):
    print(list[i])
```

Os dois retornam o mesmo resultado, mas, apesar do primeiro ser um pouco mais fácil de se escrever, o segundo é **muito** mais seguro.

Um exemplo de bom uso é o código abaixo, que retorna a concatenação da lista n:

```
n = ["Ana", "Silva"]

def concat(palavras):
    result = ""
    for i in range(0, len(palavras)):
        result = result + palavras[i]
    return result

print(concat(n))

>>> AnaSilva
```

7.4.2. Modificando cada elemento em uma lista dentro de uma função

Podemos utilizar a função range para percorrer uma lista e aplicar modificações a cada um de seus itens. O código abaixo dobra o valor de cada item da lista `n`:

```
n = [3, 5, 7]

def dobra_list(x):
    for i in range(0, len(x)):
        x[i] = x[i] * 2
    return x

print(dobra_list(n))

>>> [6, 10, 14]
```

7.4.3. Usando uma lista de listas em uma função

É possível ter uma lista formada por um conjunto de sublistas, como a lista n no código a seguir. Como percorrer os itens das sublistas?

A função `flat` que aparece no código abaixo toma uma única lista e concatena todas as sublistas que são parte dela em uma lista só.

```
n = [[1, 2, 3], [4, 5, 6, 7, 8, 9]]

def flat(lista):
    result = []
    for i in range(0, len(lista)):  # Percorre a lista
        for j in range(0, len(lista[i])):  # Percorre as sublistas
            result.append(lista[i][j])
    return result

print(flat(n))

>>> [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Repare que a função `flat` não modifica a lista `n`. Ela inicia uma lista vazia `result` e adiciona os valores a ela, retornando-a ao final. Se quisermos armazenar o resultado em uma variável nova, podemos escrever `nova_lista = flat(n)`.

8. ENTRADA E SAÍDA DE ARQUIVOS

8.1. Introdução a I/O de arquivo

Quando se quer ler informações de um arquivo no seu computador, e/ou escrever essas informações em outro arquivo, utiliza-se o processo de I/O de arquivo (I/O significa "input/output", ou "entrada/saída"). O Python tem diversas funções embutidas que tratam disso.

8.1.1. Diretório de trabalho

Ao se trabalhar com entrada e saída de arquivos podemos primeiramente determinar em qual diretório iremos realizar nossas ações. Isso é mais eficiente do que escrever o caminho completo para os arquivos toda vez que executamos uma ação. A ferramenta adequada aqui é o módulo os:

```
import os
print(os.getcwd())   # Exibe o diretório de trabalho atual

>>> 'C:\\Users\\Thiago'
```

O código acima exibe o diretório em que se está trabalhando atualmente com o Python. Se desejamos modificar este diretório para a área de trabalho, podemos escrever:

```
os.chdir('C:\\Users\\Thiago\\Desktop')
```

Repare que a separação de pastas se dá por duas barras invertidas e não uma.

8.1.2. A Função open()

Veja o código:

```
f = open("output.txt", "w")
```

Isso instrui o Python a abrir o arquivo de texto `"output.txt"` em modo de escrita (`"w"` significa write = escrever). Armazenamos o resultado dessa operação em um objeto de arquivo chamado `f`. Fazer isso abre o modo de escrita e prepara o Python para enviar dados para o arquivo. Outros modos também são possíveis nesta função.

8.1.3. Escrita de arquivo

Depois de abrir um arquivo, podemos escrever nele utilizando o método `write()`. Este método toma um argumento no formato string e, após chama-lo, você **deve** fechar o arquivo. Isso é feito chamando `my_file.close()`. Se você não fechar seu arquivo, o Python não escreverá nele corretamente. Vejamos o exemplo abaixo:

```
lista = [i ** 2 for i in range(1, 11)]

arquivo = open("output.txt", "r+")

for i in range(0, len(lista):
    arquivo.write(str(lista[i]) + "\n")

arquivo.close()
```

Primeiramente foi criado o objeto `lista`. Depois foi usado `open()` para abrir o arquivo `output.txt` (existente em nosso diretório de trabalho) atribuindo-o a `arquivo`, no modo `r+`, que permite leitura e escrita.

Percorreu-se `lista` usando `arquivo.write()` para escrever cada um de seus valores em `output.txt`. Chamou-se `str()` sobre os dados para que `write()` os aceite. A expressão `"\n"` adiciona uma nova linha depois de cada elemento da lista, para garantir que cada item aparecerá em sua própria linha. Por fim, `arquivo.close()` foi usado para fechar o arquivo.

8.1.4. Leitura

Se desejamos ler nosso arquivo criado no item anterior, fazemos isso com a função `read()`, assim:

```
arquivo = open("output.txt", "r")  # Arquivo aberto no modo leitura
print(arquivo.read())
arquivo.close()
```

8.2. Outros detalhes

8.2.1. Lendo Entre as Linhas

Se quisermos ler um arquivo linha por linha, em vez de carregar todo o arquivo de uma vez, usamos a função `readline()`.

Se abrir um arquivo e chamar `readline()` sobre o objeto arquivo, terá a primeira linha do arquivo; chamadas subsequentes de `readline()` retornarão linhas sucessivas. Veja o exemplo abaixo:

```
outro_arquivo = open("texto.txt", "w")
outro_arquivo.write("Primeira linha do arquivo!\n");
outro_arquivo.write("Segunda linha.\n");
outro_arquivo.write("Terceira linha.\n");
outro_arquivo.close()

outro_arquivo = open("text.txt", "r")
print(outro_arquivo.readline())

>>> Primeira linha do arquivo!

print(outro_arquivo.readline())

>>> Segunda linha.

print(outro_arquivo.readline())

>>> Terceira linha

outro_arquivo.close()
```

8.2.2. Utilizando with e as

Há um modo de fazer o Python fechar automaticamente os arquivos. Os objetos arquivo contêm um par especial de métodos embutidos: `__enter__()` e `__exit__()`. Quando o método`__exit__()` de um objeto arquivo é invocado, ele automaticamente fecha o arquivo. Invocamos esse método com `with` e `as`.

A sintaxe de `with` e `as` pode ser entendida assim: "**Com** objeto **como** variável, faça o que está no bloco de código recuado". O código abaixo abre o arquivo `texto.txt` no modo escrita ("`w`"), gravando (`write`) a string "`Sucesso!`":

```
with open("texto.txt", "w") as arq_txt:  # Modo escrita
    arq_txt.write("Sucesso!")

with open("texto.txt", "r") as arq_txt:  # Modo leitura
    print(arq_txt.read())

>>> Sucesso!
```

8.2.3. Verificar fechamento

Há um modo de testar se um arquivo que abrimos está fechado. Às vezes teremos muitos objetos arquivo abertos e, se não tivermos cuidado, eles não serão todos fechados. Utilizamos o atributo `closed` para isso:

```
f = open("arqv.txt")    # Abrimos um arquivo qualquer em f
f.closed                # Retorna False
f.close()               # Fecha o arquivo
f.closed                # Retorna True
```