Introdução à programação e lógica de programação

Sumário

Instalando o Python

Instalação

```
1. Acesse o link https://www.python.org/downloads/ e faça o download da versão
3 do python
2. Execute o instalador
3. Durante a instalação certifique-se de ticar a opção "Add Python 3.X to PATH"
(Adicionar Python 3.X ao PATH) na parte      inferior da tela de instalação
4. Pronto, o interpretador do Python está instalado na sua máquina
```

Validando a instalação

Existem várias formas de validar se a instalação foi com sucesso. A seguir algumas delas:

```
- Verifique no menu de aplicações instaladas (no menu iniciar) se existe um dir
etório chamado Python
- Abra o prompt de comando (cmd), digite python, aperte enter e veja se aparece
o texto (a versão do python pode ser diferente):
   *Python 3.6.2 (v3.6.2:5fd33b5, Jul  8 2017, 04:57:36) [MSC v.1900 64 bit (AM
D64)] on win32
    Type "help", "copyright", "credits" or "license" for more information.
    `>>>` _ *

- Procure por um aplicativo chamado IDLE. O IDLE é um terminal do python onde v
ocê poderá executar comandos e ver os resultados em tempo real
```

O que é o Python

Pyhton é uma linguagem de programação de propósito geral. Com ela você pode escrever desde scripts pequenos para tarefas simples, criar automação de processos, realizar processamento e análise de dados, criar sistemas web até realizar processamento de volumes de dados grandes.

Como executar programas e comandos do Python

Existem algumas formas diferentes de executar comandos ou programas escritos em python:

```
1. Criando um arquivo com extensão .py que contém código python válido e execut
ando-o.
2. Abrindo o terminal do python (executando python no prompt de comando, ou o I
DLE que vêm com a instalação do python)
```

No geral, o terminal do Python é usado para testar comandos (ou aprender a linguagem), enquanto arquivos com extensão .py são usados quando se está criando um programa propriamente dito.

Como seguir este curso

Você pode seguir este curso executando os exemplos e exercícios no terminal do python (do prompt de comando ou IDLE), mas fica aberto ao leitor a forma como deseja executar estes comandos.
Outras alternativas são: salvando arquivos com extensão .py ou abrindo o jupyter notebook e executando os comandos dentro do notebook.

Primeiro exemplo: Hello World!

In [1]:
```
print("Hello world!")
```

```
Hello world!
```

O código acima é um exemplo clássico usado no primeiro contato de uma pessoa com o mundo da programação.

Neste código foi usada uma função do Python que imprime (i.e. exibe na tela) o texto passado a esta função.

A função tem o nome de **print** e entre os **parênteses** que segue o nome da função é passado o que a função deve exibir na tela, no caso o texto **"Hello world!"**.

Mais a frente entraremos em detalhes sobre o que é exatamente uma função.

Exercício: usando a função print, imprima na tela seu nome.

Variáveis

Variáveis são espaços de memória onde são armazenados valores a serem usados em um programa.

Os valores podem ser número inteiros, números reais, caracteres, textos, etc

As variáveis têm o nome definido pelo programador. A única regra é que variáveis devem começar com uma letra ou underscore (não podem começar com números, caracteres especiais, etc)

In [2]:
```
x = 10 ## atribuindo o valor 10 à variável x
```

In [3]:
```
x = "c" ## atribuindo um caractere à variável x
```

In [4]:
```
_z = 1.34 ## atribuindo um número real à variável _z
```

Quando um valor é reatribuido à uma variável já existente, o valor anterior é descartado. Se a variável ainda não existe, ela é criada e armazenada em memória

In [5]:
```
print(x)     ## irá imprimir o último valor atribuido a x
print(_z)
```

```
c
1.34
```

Exercício: crie uma variável com o nome que desejar. Imprima esta variável. Valide se o resultado imprimido é o mesmo que você definiu

Comentários no código

Você pode adicionar comentário no código, isto é, textos que não serão executados mas que pode explicar o código ou auxiliar outro programador na leitura do código.

O caractere **#** define o começo de comentários. Tudo que vem após **#** é ignorado na hora da execução do código.

Exercício: reveja a seção anterior e identifique o que é comentário e o que não é comentário

Exemplo:

```
In [6]: x = 10 # isto é um comentário. A atribuição do valor 10 à variável x está antes do comentário então esta atribuição será executada

        # y = 30 esta linha inteira é interpretada como um comentário, então não será atribuido 30 à variável y, a linha é ignorada
```

Exercício: adicione um comentário no código que você criou no exercício anterior. Re-execute este código e veja se continua funionando da mesma forma.

Tipos de dados (Data Types)

O valor de uma variável possui um tipo.

O tipo da variável define o que ela representa (por exemplo, um texto, um número) e as operações que podem ser feitas com ela.

A linguagem define alguns tipos básicos, chamados de tipos primitivos (são os tipos mais simples possíveis e já embutidos na linguagem). A linguagem possui tipos mais complexos e o programador pode definir seus próprios tipos também, mas isto será abordado mais a frente.

Os tipos primitivos (Built-in types) são:

Numéricos

int (inteiro)

exemplos:
10
-323
3042123

In [7]:
```
x = 10
print(x)
print(type(x))
```

```
10
<class 'int'>
```

No exemplo acima usamos uma função nova, chamada ***type***. A função type exibe o tipo da variável passada a ela. Imprimimos esse resultado usando a função print.

Poderíamos também atribuir o resultado da função ***type*** a uma variável e depois imrpimi-lo, por exemplo:

In [8]:
```
x = 10
tipo_de_x = type(x)  # atribui o resultado de type(x) à variável tipo_de
_x
print(tipo_de_x)
```

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

O código acima têm o mesmo efeito do código anterior.

Exercício: defina uma variável e atribua a ela um número inteiro. Imprima o valor da variável e o tipo da variável. Valide os resultados.

float (real, conhecido como ponto flutuante)

exemplos:
1.34
4.1234
-1241243.124512312

In [9]:
```
x = 1.34
print(x)
print(type(x))
```

```
1.34
<class 'float'>
```

Exercício: defina uma variável e atribua a ela um número com casas decimais. Imprima o valor da variável e o tipo da variável. Valide os resultados.

boolean

exemplos:
True
False

In [10]:
```
v1 = False
v2 = True
print(v1)
print(type(v1))
```

```
False
<class 'bool'>
```

Exercício: defina uma variável e atribua a ela um valor booleano. Imprima o valor da variável e o tipo da variável. Valide os resultados.

list

Representam listas de valores.
exemplo: [10, 20, 1, 5, 30]

In [11]:
```
l = ['a', 'b', 'c', 'd', 'e', 'f']
print(l)
print(type(l))
```

```
['a', 'b', 'c', 'd', 'e', 'f']
<class 'list'>
```

Exercício: defina uma variável e atribua a ela uma lista contendo alguns valores inteiros. Imprima o valor da variável e o tipo da variável. Valide os resultados.

set

Representam conjuntos (coleção de valores onde cada valor só ocorre uma vez)

In [12]:
```
l = set([1,2,3,4,4])     # o valor quatro será adicionado uma única vez
l2 = {1,2,3,4,4}
print(l)
print(type(l))
print(l2)
print(type(l2))
```

```
{1, 2, 3, 4}
<class 'set'>
{1, 2, 3, 4}
<class 'set'>
```

Exercício: defina uma variável e atribua a ela um set contendo valores float. Imprima o valor da variável e o tipo da variável. Valide os resultados.

tuple (tupla)

É uma lista imutável de valores

In [13]:
```
t = (1,2,3)
print(t)
print(type(t))
```

```
(1, 2, 3)
<class 'tuple'>
```

Exercício: defina uma variável e atribua a ela uma tupla contendo strings. Imprima o valor da variável e o tipo da variável. Valide os resultados.

dict (dicionários)

Representam mapeamentos de valores. A única restrição é que o valor à esquerda do mapeamento (a chave) deve ser imutável.

In [14]:
```
d = {'azul' : 5, 'vermelho' : 3, 'preto' : 4}
print(d)
print(type(d))
```

```
{'azul': 5, 'vermelho': 3, 'preto': 4}
<class 'dict'>
```

No exemplo acima mapeamos o valor 'azul' para o valor 5, 'vermelho' para o valor 3, 'preto' para o valor 4

Exercício: defina uma variável e atribua a ela um dicionário contendo os nomes dos membros da sua família como chave e suas respectivas datas de nascimento como valores. Imprima o valor da variável e o tipo da variável. Valide os resultados.

Operações comuns para os tipos built-in

Cada tipo define um conjunto de operações que podem ser realizadas com seus valores.

Operações com tipos numéricos (int, float)

In [15]:
```
print(10 + 20)   # soma
print(10 - 20)   # subtração
print(10 * 3)    # multiplicação
print(20 / 3)    # divisão
print(20 // 3)   # divisão com resultado inteiro
print(10 % 3)    # retorna o resto de 10 dividido por 3 (módulo)
```

```
30
-10
30
6.666666666666667
6
1
```

In [16]:
```
# É possível realizar operações entre mais de um tipo numérico
x = 10 + 1.3    # operação entre int e float
print(x)
print(type(x))   # o tipo resultante de uma soma de um int com um float
será um float
```

```
11.3
<class 'float'>
```

Operações com tipo booleano

In [17]:
```
x = False
y = True
```

In [18]:
```
### Operador and
x and y   # x é verdadeiro E y é verdadeiro?
```

Out[18]: False

In [19]:
```
### Operador or
x or y   # x é verdadeiro OU y é verdadeiro?
```

Out[19]: True

In [20]:
```
### Operador not (negação)
x = not True
# aqui x deve ter o valor False
print(x)
```

```
False
```

Tabela verdade para a operação E (AND)

Exibindo valores para **p**, **q**, **p E q**

V representa True (verdadeiro)
F representa False (falso)

P	Q	P∧Q
V	V	V
V	F	F
F	V	F
F	F	F

Você pode montar a tabela verdade para operadores booleanos difrerentes, como por exemplo para o operador OU (OR) ou até mesmo para combinações dos operadores booleanos

Operações com string

In [21]:
```
# concatenação
a = 'pizza ' + 'de ' + 'pepperoni'
print(a)
```

```
pizza de pepperoni
```

In [22]:
```
# formatação de strings
a = 'pizza de %s %d %3.5f' % ('pepperoni', 10, 234231.2412351)
print(a)

d = {'sabor' : 'x', 'sabor2' : 'y'}
a = 'pizza de {sabor} e {sabor2} {z}'.format(sabor='x', sabor2='y', z=20
)
print(a)
```

```
pizza de pepperoni 10 234231.24124
pizza de x e y 20
```

In [23]:
```
##### acessando um caractere de uma string
a = 'pizza de pepperoni'
primeiro_caractere = a[0]
print("Primeiro caractere: ", primeiro_caractere)

segundo_caractere = a[1]
print("Segundo caractere: ", segundo_caractere)

primeiros_dois_caracteres = a[4:100]
print("Primeiros dois caracteres: ", primeiros_dois_caracteres)
```

```
Primeiro caractere:  p
Segundo caractere:  i
Primeiros dois caracteres:  a de pepperoni
```

In [24]:
```
# operações entre string e int
a = 'O cliente deseja pedir ' + str(10) + ' pizzas'
print(a)
```

```
O cliente deseja pedir 10 pizzas
```

operações com set

In [25]:
```
# listando as operações
print(set((1,2,3)))

#A função help exibe uma descrição do tipo que passamos para esta função
. Neste caso estamos imprimindo o help do tipo 'set'
help(set)
```

```
{1, 2, 3}
Help on class set in module builtins:

class set(object)
 |  set() -> new empty set object
 |  set(iterable) -> new set object
 |
 |  Build an unordered collection of unique elements.
 |
 |  Methods defined here:
 |
 |  __and__(self, value, /)
 |      Return self&value.
 |
 |  __contains__(...)
 |      x.__contains__(y) <==> y in x.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __iand__(self, value, /)
 |      Return self&=value.
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __ior__(self, value, /)
 |      Return self|=value.
 |
 |  __isub__(self, value, /)
 |      Return self-=value.
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __ixor__(self, value, /)
 |      Return self^=value.
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate sign
ature.
 |
 |  __or__(self, value, /)
 |      Return self|value.
 |
 |  __rand__(self, value, /)
 |      Return value&self.
```

As operações de set (abaixo) podem ser encontradas no ***help(set)*** que foi executado anteriormente.

In [26]:
```
# adicionando um elemento a um set
s1 = set([1,2,3])
s1.add(10)
print("Adicionando elemento: ", s1)

# adicionando varios elementos a um set
s1, l2 = set([1,2,3]), [3,4,5]
s1.update(l2)
print("Adicionando vários elementos: ", s1)

# união de dois sets
s1, s2 = set([1,2,3]), set([3,4,5])
s3 = s1.union(s2)
print("União de dois sets: ", s1, s3)

# intersecção de dois sets
s1, s2 = set([1,2,3]), set([3,4,5])
s3 = s1.intersection(s2)
print("Intersecção de dois sets:", s1, s3)
```

```
Adicionando elemento:  {10, 1, 2, 3}
Adicionando vários elementos:  {1, 2, 3, 4, 5}
União de dois sets:  {1, 2, 3} {1, 2, 3, 4, 5}
Intersecção de dois sets: {1, 2, 3} {3}
```

Operações com listas

In [27]:
```
# concatenação de listas
l1 = [1,2,3]
l2 = [4,5,6]
print(l1 + l2)
print(l1, l2)
#help(list)
l1.extend(l2)
print(l1)
```

```
[1, 2, 3, 4, 5, 6]
[1, 2, 3] [4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

In [28]:
```
# acessando um índice de de uma lista
l1 = [10, 20, 30]
item_0_da_lista = l1[0]
item_1_da_lista = l1[1]
item_2_da_lista = l1[2]
print(item_0_da_lista)
print(item_1_da_lista)
print(item_2_da_lista)
```

```
10
20
30
```

In [29]:
```
# acessando uma 'fatia' da lista
l1 = [0, 10, 20, 30, 40, 50, 60]
items_4_a_6 = l1[4:7] + l1[1:2]
print(items_4_a_6)
```

```
[40, 50, 60, 10]
```

In [30]:
```
# adicionando um elemento ao final da lista
l1 = [0, 10, 20, 30]
l1.append(200)
print(l1)
```

```
[0, 10, 20, 30, 200]
```

In [31]:
```
# acessando um elemento a partir do fim da lista
l1 = [0, 10, 20, 30, 40, 50, 60]
ultimo_item = l1[-1]
penultimo_item = l1[-8:-2:2]
print("Ultimo item: ", ultimo_item)
print("Penultimo item: ", penultimo_item)
```

```
Ultimo item:  60
Penultimo item:  [0, 20, 40]
```

operações com dicionários

In [32]:
```
# acessando um elemento do dicionário
d = {'azul' : 5, 'preto' : 10, 0 : 100, 1 : 200}
contagem_azul = d[0]
print(contagem_azul)
```

```
100
```

In [33]:
```
# adicionando um valor a um dicionário
d['branco'] = 10
print(d)
```

```
{'azul': 5, 'preto': 10, 0: 100, 1: 200, 'branco': 10}
```

In [34]:
```
# removendo um valor de um dicionário
print("Antes: ", d)
del d['branco']
print("Depois: ", d)
```

```
Antes:  {'azul': 5, 'preto': 10, 0: 100, 1: 200, 'branco': 10}
Depois:  {'azul': 5, 'preto': 10, 0: 100, 1: 200}
```

Funções

As funções são sequencias de operações com a finalidade de executar uma determinada tarefa.

No geral estas funções recebem valores de entrada e retornam valores de saída.

A linguagem Python fornece algumas funções já implementadas (por exemplo o print(...) é uma função definida pela linguagem), mas o programador pode também definir suas funções.

A grande vantagem de definir uma sequencia de código como uma função é a capacidade de reutilizar o código desta função sem precisar reescrevê-lo. Basta executar a função pelo nome dado a ela. Reusabilidade de código é um fator muito importante.

Por exemplo, upodemos criar uma função que calcula a raiz quadrada de um número:

In [35]:
```
def quadrado(x):          ## def diz que definiremos uma função. Em sequencia vem o nome da função que escolhemos e entre parenteses os parâmetros que a função recebe
    resultado = x*x       ## aqui começa as operações a serem executadas na função
    return resultado      ## aqui retornamos o resultado para quem usará esta função
```

Usando esta função:

In [36]:
```
r = quadrado(10)   # aqui executamos a função e guardamos o resultado (o retorno da função) na variável r
print(r)
```

```
100
```

Uma função pode não conter argumentos e/ou não retornar nada. Por exemplo, esta função não retorna nada:

In [37]:
```
def imprime_quantidade(quantidade):
    print("A quantidade de produtos é %d" % (quantidade,))
```

In [38]:
```
imprime_quantidade(231)
imprime_quantidade(30)
imprime_quantidade(50)
```

```
A quantidade de produtos é 231
A quantidade de produtos é 30
A quantidade de produtos é 50
```

Esta função não recebe nenhum argumento e não retorna nada:

In [39]:
```
def imprime_título():
    print("Curso de lógica de programação")
```

Funções built-in

O python dispõe de algumas funções já implementadas.

Alguns exemplos são:

- len: retorna a quantidade de itens em uma lista
- sum: recebe uma lista e retorna a soma da lista
- map: aplica uma função para cada elemento de uma lista
- help: mostra os métodos e propriedades de um tipo
- max: retorna o máximo de uma lista
- min: retorna o mínimo de uma lista
- print: imprime valores na tela
- ...

https://docs.python.org/3.2/library/functions.html (https://docs.python.org/3.2/library/functions.html)

Abaixo alguns exemplos do uso de funções do python:

In [40]:
```
l = [1,2,3,4]
print(len(l))   # imprimindo o resultado da função len()
print(sum(l))   # função sum
print(max(l))   # função max
print(min(l))   # função min
```

```
4
10
4
1
```

Estruturas de controle

Estrutura if

Avalia uma condição e executa um ou mais comandos caso esta condição seja verdade

Uma condição é um teste (ou uma avaliação) que fazemos. Por exemplo, no meio do código de um software de estoque, poderá ser necessário avaliar se existe estoque de um dado produto antes de executar uma ação (por exemplo, vendê-lo ao cliente).

Neste caso, se temos a quantidade de produtos em estoque em uma variável chamada ***numero_de_produtos***, podemos avaliar se o valor desta variável é maior do que 0.

Exemplo:

In [41]:
```
numero_de_produtos = 100
venda_concluida = True

# valida se temos estoque
if numero_de_produtos > 0:
    print("Temos estoque")

x = "abc"

#valida se o estoque é maior ou igual a zero
# (por exemplo se o núero de produtos é consistente, considerando que um valor negativo é uma inconsistência no sistema)
if numero_de_produtos >= 0:
    print("O valor de numero_de_produtos é consistente")

if numero_de_produtos < 0:
    print("O Valor de numero_de_produtos é inconsistente")

if numero_de_produtos <= 0:
    print("O valor de numero_de_produtos é menor ou igual a zero")

if numero_de_produtos == 0:
    print("Não temos estoque")

## decrementa o número de produtos
## aqui avaliamos duas condições usando o operador and
## avaliamos se o número de produtos é maior ou igual a zero E se a venda foi concluida
## caso sim, subtraimos 1 do valor de numero_de_produtos
if numero_de_produtos>0 and venda_concluida:
    numero_de_produtos = numero_de_produtos - 1

print("Qtd atual de produtos=", numero_de_produtos)
```

```
Temos estoque
O valor de numero_de_produtos é consistente
Qtd atual de produtos= 99
```

Quando a condição de um if é avaliada, e seu valor é True, o bloco de código executado é o bloco identado que segue a condição (ou seja, todas as linhas de código com um espaçamento em relação à lateral esquerda do editor)

Por exemplo:

In [42]:
```
x = 10
if x > 0:
    print("Esta linha será executada se x > 0")     # faz parte do bloco
if
    print("Esta linha também será executada se x > 0") # faz parte do bl
oco if
    print("Esta linha também será executada") # faz parte do bloco if
print("Esta linha já está fora do bloco do if") # não faz parte do bloco
if
print("Repare no código que o espaçamento (identação desta linha) em rel
ação à lateral esquerda é menor do que a identação dentro do bloco if")

if x < 0: #iniciando outro bloco if
    print("Esta linha de código não será executada já que x é maior que
zero (a avaliação do if resultou em False)") # faz parte do segundo bloc
o if
```

```
Esta linha será executada se x > 0
Esta linha também será executada se x > 0
Esta linha também será executada
Esta linha já está fora do bloco do if
Repare no código que o espaçamento (identação desta linha) em relação à l
ateral esquerda é menor do que a identação dentro do bloco if
```

Podemos opcionalmente testar várias condições if em sequencia. Esta estrutura (que é parte do if) é chamada de else if, e no python é chamada de **elif**.

A condição no **elif** é testada somente se o if anterior for false.

Por exemplo:

In [43]:
```
x = 10
if x>20:
    print("X é maior do que 20")
elif x>5:
    print("X é maior do que 5")
```

```
X é maior do que 5
```

Vários elif podem ser definidos em sequencia e as avaliações do bloco if/elif/elif... é abortada quando a primeira condição for avaliada como True

Exemplo:

In [44]:
```
x = 10
if x>20:
    print("X é maior do que 20")   # não será executado porque x não é m
aior do que 20
elif x>15:
    print("X é maior do que 15")   # não será executado porque x não é m
aior do que 15
elif x>10:
    print("X é maior do que 10")   # não será executado porque x não é m
aior do que 10
elif x>5:
    print("X é maior do que 5")    # será executado porque x é maior do
que 5. O bloco if/elif termina aqui, já que avaliou como True
elif x>0:                          # não será executado porque uma condi
ção anterior já foi avaliada como True
    print("X é maior do que 0")
```

```
X é maior do que 5
```

Estrutura while

In [45]:
```
import pprint

def cor_coincide(produto, cor):
    return produto['cor'] == cor

def filtra_produtos_com_base_em_cor(produtos, cor):
    qtd_produtos = len(produtos)    # obtém a quantidade de produtos na
lista
    indice_atual = 0                # inicializa um contador para percor
rer a lista (será o índice que vamos acessar)
    filtrados = []                  # a lista de items da cor que deseja
mos, inicializamos vazia
    while indice_atual < qtd_produtos:     # loop while (enquanto o indi
ce que estamos for menor do que a qtd de produtos)
        produto = produtos[indice_atual]   # obtem o produto do indice a
tual
        if cor_coincide(produto, cor):                 # valida se a cor
do produto é a cor que desejamos
            filtrados.append(produto)             # adiciona o produto d
a cor desejada ao final da lista
        indice_atual = indice_atual + 1       # incrementa o contador pa
ra validarmos a próximo item
    return filtrados                # retorna o resultado

produtos = [
    {'cor' : 'azul',
    'tamanho' : 'M'},
    {'cor' : 'azul',
    'tamanho' : 'P'},
    {'cor' : 'verde',
    'tamanho' : 'S'},
    {'cor' : 'azul',
    'tamanho' : 'G'},
    {'cor' : 'verde',
    'tamanho' : 'G'},
    {'cor' : 'marrom',
    'tamanho' : 'G'}]

produtos_verdes = filtra_produtos_com_base_em_cor(produtos, 'verde')
produtos_azuis = filtra_produtos_com_base_em_cor(produtos, 'azul')
print("Produtos filtrados: ")
pprint.pprint(produtos_verdes)
pprint.pprint(produtos_azuis)
```

```
Produtos filtrados:
[{'cor': 'verde', 'tamanho': 'S'}, {'cor': 'verde', 'tamanho': 'G'}]
[{'cor': 'azul', 'tamanho': 'M'},
 {'cor': 'azul', 'tamanho': 'P'},
 {'cor': 'azul', 'tamanho': 'G'}]
```

Estrutura for

A estrutura for simplifica o while quando precisamos iterar sobre uma lista.

Pordemos reescrever a função substituindo o while pelo for filtra_produtos_com_base_em_cor:

In [46]:
```
def filtra_produtos_com_base_em_cor(produtos, cor):
    filtrados = []
    for produto in produtos:     # loop for (para cada produto em produt
os faça:)
        if cor_coincide(produto, cor):
            filtrados.append(produto)
    return filtrados
```

In [47]:
```
produtos_azuis = filtra_produtos_com_base_em_cor(produtos, 'azul')
pprint.pprint(produtos_azuis)
```

```
[{'cor': 'azul', 'tamanho': 'M'},
 {'cor': 'azul', 'tamanho': 'P'},
 {'cor': 'azul', 'tamanho': 'G'}]
```