LÓGICA DE PROGRAMAÇÃO EM PYTHON

POR GABRIEL L C SELOW

LÓGICA DE PROGRAMAÇÃO EM PYTHON

POR GABRIEL L C SELOW

POR QUE APRENDER A PROGRAMAR?

Programação é a forma como nos comunicamos com o computador, através de regras predefinidas. Quando programamos, visamos resolver uma problemática, por meio de uma série de comandos. Assim, desenvolvemos o raciocínio lógico e a capacidade de solucionar problemas, tanto computacionais quanto cotidianos.

A indústria de TI é próspera e tende a continuar em ascensão. Em meios às altas taxas de desemprego, as vagas para programadores são crescentes em todo o Brasil. Dessa forma, nota-se a importância da programação não só como hobby ou como uma forma de desenvolver o raciocínio lógico, mas como opção de carreira. Uma carreira muito valorizada, necessitada e com excelentes médias salariais.

Portanto, como a tecnologia está presente em diversas esferas, aprender a lógica de programação nos possibilita entender e interpretar o mundo de uma forma totalmente inédita.

SUMÁRIO

CAP 1: INSTALAÇÃO DO PYTHON

CAP 2: INTRODUÇÃO

CAP 3: ALGORITMOS E FLUXOGRAMA

CAP 4: ENTRADA E SAÍDA DE DADOS, TIPOS DE DADOS, VARIÁVEIS E OPERADORES ARITMÉTICOS

CAP 5: INDENTAÇÃO, ESTRUTURAS DE DECISÃO, OPERADORES RELACIONAIS E OPERADORES LÓGICOS

CAP 6: LISTAS, TUPLAS E DICIONÁRIOS

CAP 7: ESTRUTURAS DE REPETIÇÃO

CAP 8: FUNÇÕES E BIBLIOTECAS

CAP 9: FUNDAMENTOS DE POO

1.
INSTALAÇÃO DO PYTHON

Embora o nome deste capítulo seja “instalação do python” não será necessário a instalação desta linguagem de programação para acompanhar e executar os exemplos contidos neste livro.

Portanto, a alternativa que utilizaremos será um interpretador online de python, com esta ferramenta é possível executar programas em python direto do nosso browser. Podemos achar um interpretador de python online apenas pesquisando “interpretador de python online” no google, ou utilizar o QR code abaixo, que contém o link de um interpretador online de python.

2.
INTRODUÇÃO

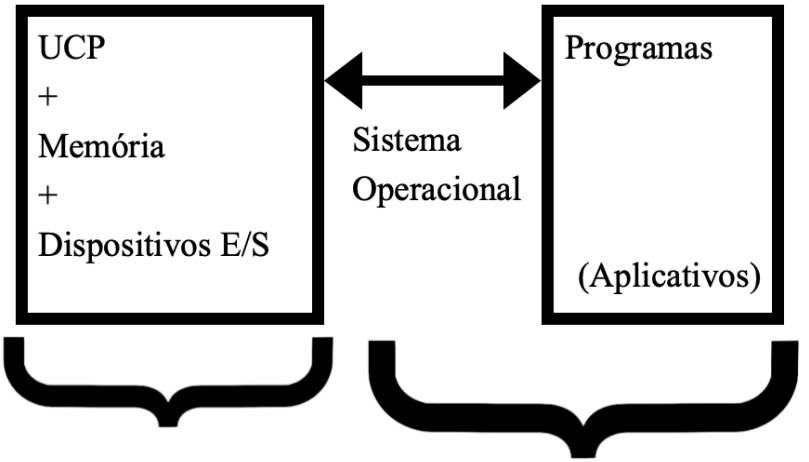
Para podermos entender a lógica de programação é necessário a compreensão de determinados conceitos. Sendo assim, neste primeiro capítulo iremos revisar conteúdos que,embora básicos, nos ajudarão muito.

Começaremos com os conceitos de software e hardware que, juntos, formam o computador como conhecemos. O hardware é a parte física do computador, formado pela UCP, pela memória e por dispositivos de entrada e saída.

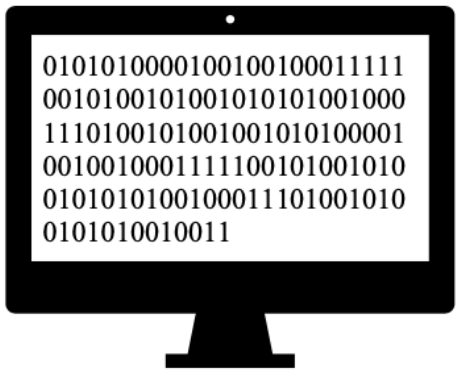
A UCP, unidade central de processamento, é responsável por executar os programas que estão armazenados na memória. Já a memória é responsável por armazenar as informações, programas e dados, que serão manipuladas pela máquina. Por fim, tem-se os dispositivos de entrada e saída, que são a forma como o computador se comunica com meio externo, como por exemplo o teclado, mouse e o monitor.

O software é a parte não física do computador, ou seja, o sistema operacional e os programas. O sistema operacional é o responsável por fazer a comunicação entre o hardware e os outros programas. Alguns exemplos de sistemas operacionais são: Mac OS, Windows, Linux e Android.

Na imagem abaixo nota-se a relação entre software e hardware:



O computador é um dispositivo que processa informações, das quais um bit é a menor parcela possível de ser processada. Na prática, o bit é um valor binário, ou seja, assume valores de 0 ou 1, normalmente entendidos como ligado ou desligado, de forma que vários bits juntos codificam informações, através do chamado código de máquina. De uma forma mais simples, o código de máquina é quando vemos em um filme alguém hackeando e aparecem várias zeros e uns na tela, como na imagem a seguir:



Para facilitar a leitura, aglomeramos os bits em grupos, de forma que cada conjunto de 8 bits é chamado de byte. Um kB (kilobyte) equivale a 1024 bytes, um MB (megabytes) a 1024kilobytes, um GB (gigabyte) a 1024 megabytes e assim por diante.

Foi citado anteriormente que bits são números binários, mas como funciona o sistema numérico binário? Cotidianamente, usamos o sistema decimal, que possui o número 10 como base, utilizando os 10 algarismos que vão de 0 a 9. Já o sistema binário utiliza apenas dois algarismos: o 0 e o 1. No sistema binário os números são escritos a partir de potencias de dois, como no exemplo abaixo:

Essa transformação é feita da seguinte forma:
1x(2^4)+1x(2^3)+0x(2^2)+1x(2^1)+1x(2^0) = 16 + 8 + 0 + 2 + 1 = 27

Por fim, temos também o sistema hexadecimal, que é composto por 16 algarismos que vão de 0 a F, sendo que A, B, C, D, E e F substituem, respectivamente, 10,11,12,13,14 e 15.

Podemos perceber na tabela abaixo a equivalência entre números binários, decimais e hexadecimais.

BINÁRIO	DECIMAL	HEXADECIMAL
0	0	0
1	1	1
10	2	2
11	3	3
100	4	4
101	5	5
110	6	6
111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Como já citado, uma linguagem de programação é a forma como nos comunicamos com o computador. Essas linguagens podem ser mais ou menos parecidas com a linguagem humana. As linguagens de alto nível são as mais parecidas e as de baixo nível, as menos.

As linguagens de alto nível são mais simples e fáceis de aprender, o que as torna mais baratas. Porém, por serem

mais parecidas com a linguagem humana, são menos eficientes se forem analisadas em relação ao aproveitamento do hardware disponível. Alguns exemplos são: Python, JS (JavaScript), entre outras.

Já as linguagens de baixo nível são mais parecidas com a linguagem de máquina. Por serem mais complexas para o entendimento humano, elevam o custo da mão-de-obra e o tempo de solução de problemas. Contudo, aproveitam melhor o hardware disponível e, por isso, são muito utilizadas quando o hardware é limitado. O principal exemplo de linguagem de baixo nível é o Assembly.

Mas se o computador só entende linguagem de máquina e nós programamos em diversas outras linguagens, como o computador nos entende? O responsável por fazer essa tradução é o compilador ou o interpretador. Dessa forma, o compilador nada mais é do que um tradutor que, geralmente, traduz uma linguagem de alto nível em uma linguagem de baixo nível.

3.
ALGORITMOS E FLUXOGRAMA

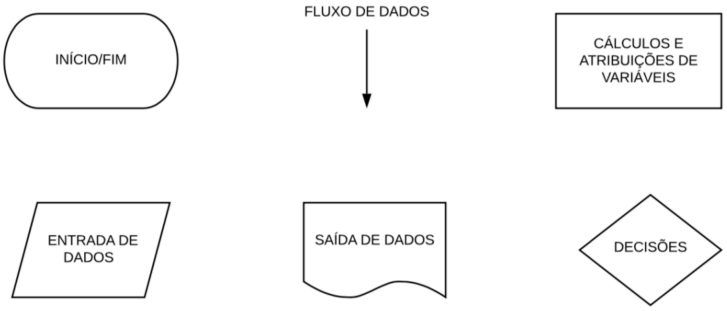
Neste capítulo, estudaremos algoritmos, principalmente, na forma de fluxograma. Mas, afinal, o que é um algoritmo? Um algoritmo é o passo a passo para a realização de uma tarefa, como trocar uma lâmpada:

TROCAR LÂMPADA

- Pegar escada
- Posicionar a escada em baixo da lâmpada
- buscar lâmpada nova
- Subir na escada
- Retirar lâmpada velha
- Colocar lâmpada nova

Os algoritmos podem ser divididos em 3 tipos. O primeiro deles é a descrição narrativa, retratada pelo exemplo acima; o segundo é o fluxograma, que será aprofundado a seguir; e o terceiro é o pseudocódigo, também chamado de Portugol. O fluxograma é a representação gráfica das etapas de um algoritmo. Logo, é necessário conhecer os seus principais

símbolos, para que se possa entender algoritmos mais complexos.A figura abaixo mostra esses símbolos e suas respectivas funções:



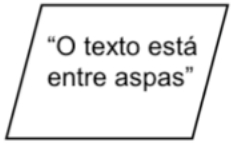
O início e o fim são representados pelo mesmo símbolo. Por isso, é necessário que escrevamos dentro dele o que ele está indicando. Já a flecha aponta o fluxo de dados, ou seja, ela indica a ordem em que cada ação está sendo executada. Dessa forma, se seguirmos o sentido da flecha, podemos ver a sequência lógica do algoritmo.

O retângulo é o símbolo usado para cálculos e atribuições. Os cálculos em questão são cálculos matemáticos. Já as atribuições são valores dados a uma **variável**, como quando igualamos, por exemplo, num=0.

Uma variável é um nome dado a um espaço na memória, onde será guardado um determinado valor. No exemplo ao lado, o nome da nossa variável é “num”, e a estamos inicializando, igualando-a a zero.

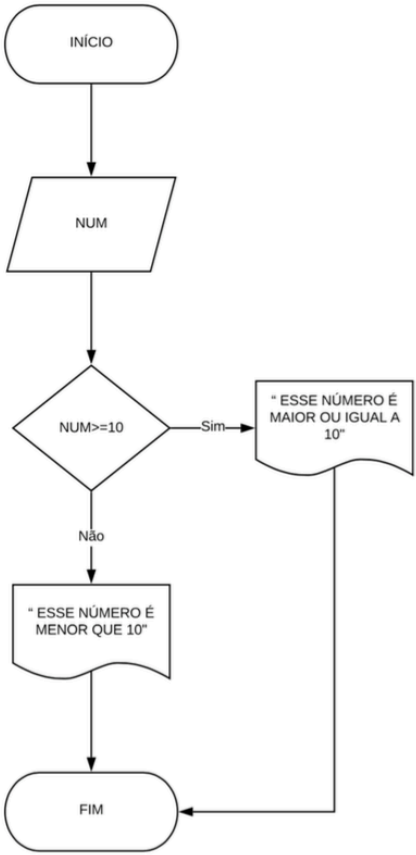
Para representar a entrada de dados, utilizamos um paralelogramo. Dentro da figura, colocamos o nome de uma ou mais variáveis, separando-as por vírgulas. Essas variáveis indicam lugares na memória onde serão armazenadas as informações advindas do teclado.

A saída é simbolizada por um retângulo com uma “ondinha” embaixo. Dentro dessa figura é colocado um texto ou o nome de uma variável que serão expressos na tela. Se temos como intuito imprimir na tela um texto, colocamo-lo entre aspas, como se nota abaixo:



Agora, se temos como objetivo imprimir os dados que estão contidos em uma variável, colocamos apenas os nomes das variáveis, sem aspas.

Por fim, existe o losango. Esse símbolo representa uma tomada de decisão. Dentro dele é colocada uma sentença que admite 2 respostas (sim ou não) e, dependendo da resposta, seguimos um caminho diferente. Por exemplo, se quisermos saber se um número é maior ou igual a 10, escreveremos a seguinte sentença: **numero>=10**. Abaixo temos o exemplo em um fluxograma:

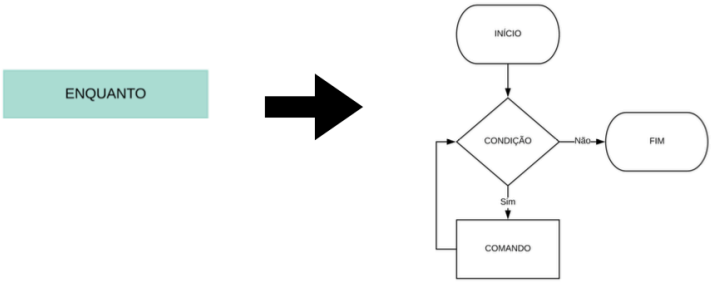


Outro ponto importante dentro desse conteúdo são as estruturas de repetição, ou também chamadas de laços de repetição. Essas tem como objetivo realizar tarefas repetidas vezes com base em uma condição.

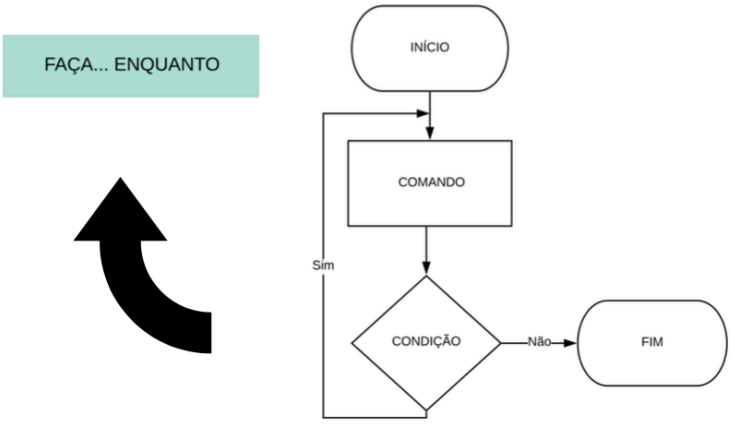
Por exemplo, se objetivamos imprimir os números de 1 a 100, podemos utilizar uma estrutura de repetição e assim economizar código, tornando-o mais limpo.

Podemos dividir as estruturas de repetição em 3:ENQUANTO, FAÇA...ENQUANTO e PARA. Na primeira delas, uma condição é verificada. Se essa for verdadeira, o bloco de código, ou seja, o comando que está dentro do

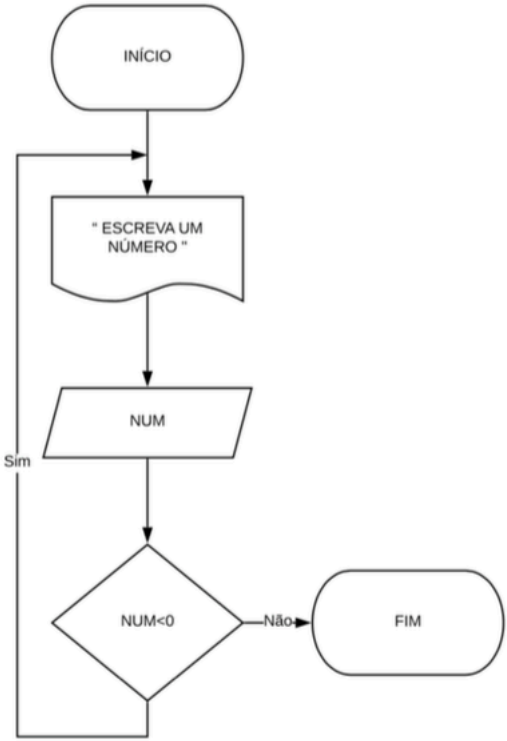
laço, será executado e assim por diante. Como no exemplo abaixo:



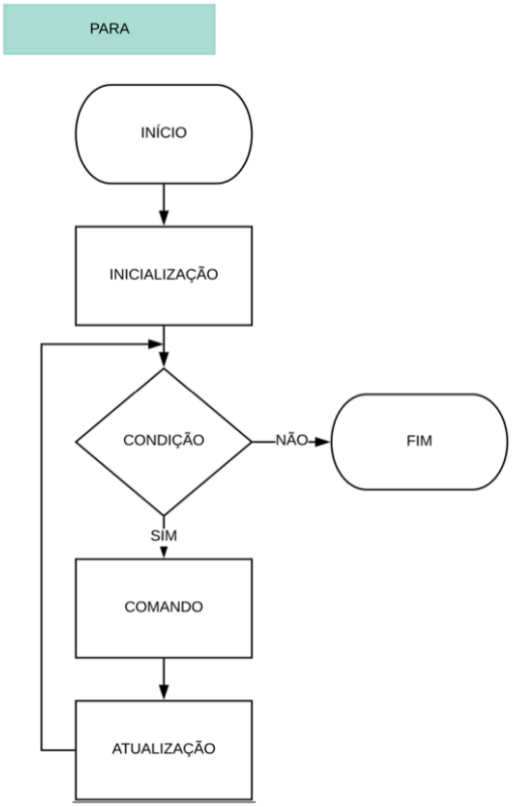
A segunda é muito semelhante à primeira, sendo que a única diferença incide em, quando usamos FAÇA... ENQUANTO, o comando é executado antes de a condição ser verificada. Caso seja verdadeira, o laço de repetição será executado. Segue um exemplo:



Essa estrutura é muito utilizada para a validação de uma entrada, através do estabelecimento de uma condição. Por exemplo, no caso de um progama que precisasse que o usuário escrevesse um número maior que 0 e, enquanto essa condição não fosse atendida, ele continuaria pedindo um número, conforme o fluxograma a seguir:

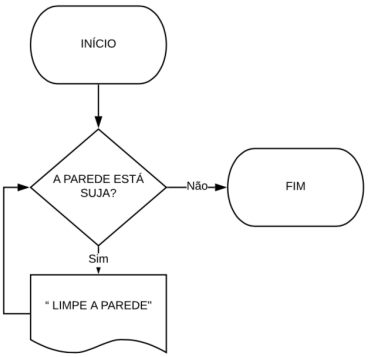


Por último, temos a estrutura PARA. Essa é utilizada para realizar uma tarefa por um número exato de vezes. Abaixo nota-se um exemplo:

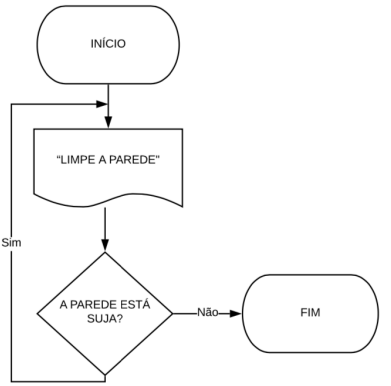


Para facilitar o entendimento dos laços de repetição, resolveremos um mesmo problema com as 3 estruturas. No exemplo, uma parede está suja e queremos limpá-la. Para as estruturas ENQUANTO e FAÇA...ENQUANTO criaremos algoritmos que continuarão a limpar a parede enquanto ela ainda estiver suja. Já para o comando PARA, o algoritmo serão programados para limpar a parede por um número exato de vezes. A seguir temos os algoritmos:

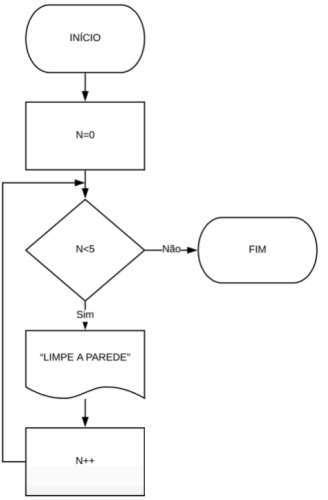
ENQUANTO



FAÇA... ENQUANTO



PARA



4.

ENTRADA E SAÍDA DE DADOS, TIPOS DE DADOS, VARIÁVEIS E OPERADORES ARITMÉTICOS

Neste capítulo, iremos começar a aprender Python. Os tópicos que serão abordados são: entrada e saída de dados, tipos de dados e variáveis e operadores aritméticos.

Começaremos com o comando print(), responsável pela saída de dados. Esse comando é utilizado para escrever algo na tela, podendo ser um texto ou o conteúdo de uma variável. Quando queremos escrever um texto, devemos colocá-lo entre aspas, pois, caso não sejam colocadas, o computador entende o texto como se fossem variáveis. O programa abaixo é um exemplo muito tradicional do primeiro código em uma nova linguagem de programação, ele é chamado de “Olá, mundo” e tem como objetivo escrever na tela a frase “Olá, mundo”.

```
print("Olá Mundo”)
```

Acima, temos um código simples, porém, mais adiante, escreveremos códigos mais complexos e explicá-los dentro deles mesmos pode ser interessante, isso se chama comentário. Ou seja, um comentário é um pedaço do código que serve apenas para o entendimento de quem está lendo, mas que não será executado. Os comentários podem ser expressos das seguintes formas: logo após o símbolo # (jogo

da velha), usado para comentar apenas uma linha, ou entre o símbolo ''' e '''(três aspas simples), utilizado para comentar diversas linhas. Abaixo, nota-se a utilização de comentários:

Outro comando muito importante é o comando de

```
print("Olá Mundo”)
#comentario de uma unica linha
''' comentario de uma ou
      múltiplas linhas '''
```

entrada. Para receber alguma informação digitada pelo usuário, usamos o comando input( ). Neste comando, além de recebermos uma informação do usuário, podemos escrever um texto na tela. Devemos escrever o texto que será impresso na tela entre aspas. É necessário também que igualemos esse comando a alguma variável, para que a informação recebida do usuário seja armazenada na memória. Abaixo, receberemos o nome do usuário através do comando de entrada input( ) e o exibiremos utilizando o comando print( ).

```
nome=input(“escreva seu nome”)
print(“seu nome e”, nome)
```

Existem diversas formas de escrever variáveis junto ao texto -- abordaremos 3. A primeira é a que vimos no exemplo anterior, que consiste em escrever o texto que será impresso na tela entre aspas e separá-lo por vírgula do nome da variável que contém a informação que aparecerá junto a ele. A segunda opção é utilizar um marcador, que é quando

colocamos ”%s ” para marcar uma string, por exemplo. Quando fazemos uma marcação, devemos indicar a variável após o texto por uma % e, em seguida, o nome da variável. As marcações variam de acordo com o tipo da variável, abordaremos isso melhor a seguir.A terceira e última forma é a que usa o método format. Com ele, o programa coloca o conteúdo da variável entre as chaves indicadas no meio do texto.

```
nome=input(“escreva seu nome”)
print(“seu nome e”, nome)
print("O nome informado foi: %s" %nome)
print("O nome informado foi: {}”.format(nome))
```

A variável é um nome dado a um espaço alocado na memória, onde será guardado um determinado valor; esse valor denomina-se dado. Em Python, trabalhamos com 7 tipos principais de dados, colocados na tabela abaixo:

TIPO DE DADOS	BREVE DESCRIÇÃO
int	para números inteiros
str	para conjunto de caracteres
bool	armazena True ou False
float	para números com virgula
list	para agrupar um conjunto de elementos
tupla	semelhante ao tipo list, porém, imutável

dic	para agrupar elementos que serão recuperados por uma chave

A classificação dos dados serve para que o computador não os interprete erroneamente.

O tipo int é utilizado para números inteiros (positivos e negativos que não possuem casas decimais). Quando utilizamos esse tipo de variável o marcador que deve ser utilizado é o %d.

Já o str é usado para trabalharmos com um conjunto de caracteres.´ importante salientar que esse conjunto pode possuir apenas um elemento. O marcador correspondente para ele é o %s.

O bool é se emprega para valores booleanos; ou seja, quando se pode assumir apenas dois valores: o true ou false.

Temos ainda o float, que é usado para números com vírgula (possui casas decimais) e seu marcador é o%f.

Listas, tuplas e dicionários serão vistos nos próximos capítulos.

Para criarmos uma variável, basta que a iniciemos, atribuindo um valor ou dando um comando de entrada de dados. No exemplo abaixo, criaram-se variáveis diferentes de informações sobre um determinado indivíduo.

```
nome=“Davi"
sexo=‘M'
idade=18
altura=1.55
casado=true
```

Caso tenhamos como intuito ler um tipo específico de dado, podemos incrementar o comando input. Por exemplo, se quisermos ler um número inteiro, faremos assim:

```
nome=int(input(“Digite um numero inteiro\n”))
```

Notamos também a necessidade de nos aprofundarmos nos comandos de atribuição. Esses são simples, mas se não forem bem compreendidos podem prejudicar nosso desenvolvimento na programação. Basicamente, o comando de atribuição é o “= “. A variável que vai receber o valor sempre fica a esquerda do símbolo, e o valor, a direita. Se quisermos, por exemplo, adicionar 1 a uma variável e guardar esse novo valor na mesma variável, faremos da seguinte forma:

variavel = variavel + 1

Por fim, discutiremos o que são operadores aritméticos. Eles nada mais são do que as operações entre números. Podemos realizar a soma, subtração, multiplicação ou divisão usando, respectivamente, os símbolos +, - , * e / . Além das operações fundamentais da matemática, temos também o operador %, que nos dá o resto de uma divisão, e o operador **, que representa uma potência. O operador % foi utilizado no exemplo a seguir para calcularmos o resto da divisão entre 7 e 2. O resto é igual a 1, como se nota abaixo:

```
a=2
b=7
c=b%a
print(c) //c tem o valor de 1
```

Quando utilizamos o operador “+” (soma) para “somarmos” dois textos, esse vira um operador de concatenação, junção de textos. Por exemplo, quando concatenamos os textos “ola ”+”mundo”=“ola mundo”.

5.
INDENTAÇÃO, ESTRUTURAS DE DECISÃO, OPERADORES RELACIONAIS E OPERADORES LÓGICOS

Neste capítulo, os temas que abordaremos são: estruturas de decisão, operadores relacionais e operadores lógicos.

Estruturas de decisão eram representadas pelo losangono fluxograma e possibilitavam ao programa escolher entre dois caminhos diferentes, de acordo com a sentença condicional. Em Python é muito semelhante, mas, ao invés da representação gráfica, utilizamos uma representação escrita. Assim, podemos fazer com que o programa execute algum comando apenas se uma determinada situação ocorrer.

Para escrevermos as sentenças, ou seja, as condições, fazemos a utilização dos chamados operadores relacionais, que se resumem nos símbolos apresentados na tabela abaixo:

Operadores	**Explicação**
== (igual)	Utilizado para verificar se dois valores são iguais
!= (diferente)	Utilizado para verificar se dois valores são diferentes
< (menor que)	Utilizado para ver se um valor é menor que outro
> (maior que)	Utilizado para ver se um valor é maior que outro
<= (menor ou igual)	Utilizado para ver se um valor é menor ou igual a outro
>= (maior ou igual)	Utilizado para ver se um valor é maior ou igual a outro

Voltando às estruturas de decisão, temos, primeiramente, o comando if. Sua estrutura é:

```
if condição:
        comando(s)
```

É possível notar que na linha seguinte ao comando if, demos um espaçamento, chamamos isso de indentação. A indentação, em Python, é responsável por delimitar blocos de códigos, ou seja, tudo que vier após o comando if e tiver um espaçamento da margem faz parte dos comandos que serão executados, caso a condição seja verdadeira.

Em outras linguagens, os blocos são delimitados por chaves. Nesses casos, a indentação é uma questão estética, mas, mesmo assim, muito importante, já que facilita a leitura do código. Porém, em Python, a indentação vai além de uma questão estética. Tudo que está rente à margem faz parte do primeiro nível hierárquico; já o que está com uma tabulação ou 4 espaços representa o segundo nível hierárquico; o que tiver 2 tabulações ou 8 espaços está dentro do terceiro nível hierárquico e assim por diante.

A segunda estrutura de decisão é composta pelo comando if seguido pelo comando else. Nessa estrutura existe uma condição e, caso ela seja atendida, o primeiro bloco de códigos será executado. Caso contrário, será executado o bloco de código que vem após o comando else, como é possível observar abaixo:

```
if condição:
        comando(s)
else:
        comando(s)
```

Suponhamos que desejemos criar um algoritmo para auxiliar alguém a atravessar a rua. Nesse programa, frente a esta suposta situação, o usuário digitará 1 se tiver carros na rua e 2 se a rua estiver livre. Dependendo da resposta, o programa escreverá na tela para o usuário atravessar, ou não, a rua:

```
num=input("digite 1 se tiver carros na rua e 2 se a rua estiver livre”)
if num==1:
print("nao atravesse a rua") else:
print("atravesse a rua")
```

Para ajudar a compreender melhor o conteúdo, faremos outro exemplo no qual o usuário poderá escolher se ele deseja realizar uma soma ou subtração entre dois números. No exemplo, podemos notar que temos a palavra int antes do comando input. Esse comando transforma uma string em um número inteiro e pode ser utilizado em diversas outras situações que objetivam transformar algum tipo de dado em inteiro.

```
num1=int(input("digite dois numeros\n”))
num2=int(input())
op=input("digite a oprecao que deseja realizar\n")
if op=='+':
        print(num1+num2)
else:
print(num1-num2)
```

Porém, é muito comum que exista a necessidade de mais do que apenas dois caminhos a serem seguidos pelo programa. Por exemplo, se, diferentemente do programa acima, que realizava apenas soma e subtração, quiséssemos fazer um programa que utilizasse as 4 operações, como uma calculadora, teríamos que utilizar a estrutura if...elif...else, como notamos a seguir:

```
num1=int(input("digite dois numeros\n”))
num2=int(input())
op=input("digite a oprecao que deseja realizar\n”)
if op=='+':
        print(num1+num2)
elif op=='-':
        print(num1-num2)
elif op=='*':
        print(num1*num2)
elif op=='/':
        print(num1/num2)
else:
        print("esta operacao nao existe")
```

Por fim, discutiremos os operadores lógicos, que são basicamente os operadores E e OU e que indicam apenas dois valores -- o valor verdadeiro e o falso.

O primeiro deles funciona da seguinte maneira: caso duas afirmações sejam verdadeiras, ele resultará em um afirmação verdadeira; caso contrário, o resultado será falso.

Já o operador OU resulta em verdadeiro se alguma das afirmações for verdadeira. Ou seja, para que a resposta seja verdadeira basta que pelo menos uma das afirmativas também seja, e para a resposta ser falsa é necessário que ambas as afirmativas sejam falsas.

Abaixo, notam-se alguns exemplos que envolvem os 2 operadores:

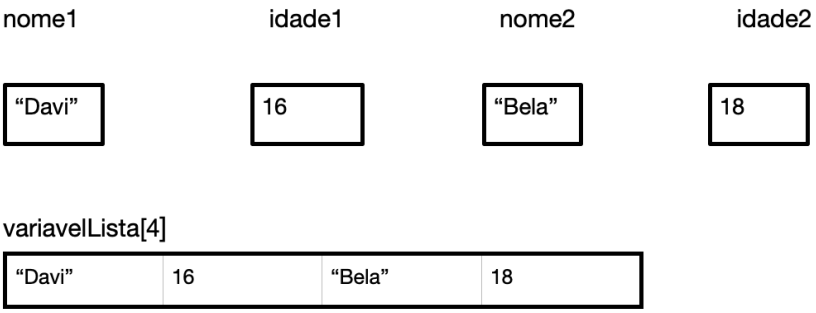
Escreva V para verdadeiro e F para falso.
(V) 8 e 9 são menores que 10
(V) 11 ou 13 são maiores que 12
(V) 11>12 ou 13>12

Em Python, o operador E é simbolizado por AND e o operador OU, por OR. Abaixo, temos um exemplo da utilização destes operadores lógicos na programação. Neste programa, temos como objetivo verificar se um indivíduo está na média nacional de altura, sabendo que a média está entre 150 e 180cm.

```
altura=int(input("escreva sua altura”))
if altura>150 and altura<180:
        print("voce esta dentro da altura media nacional”)
else:
        print("voce nao esta dentro da altura media nacional")
```

6.
LISTAS, TUPLAS E DICIONÁRIOS

Listas, que serão abordadas neste capítulo, são, de forma simplificada , variáveis “especiais” capazes de guardar várias variáveis “dentro” delas. Por exemplo, se tivéssemos 5 variáveis e quiséssemos armazená-las em uma única variável do tipo lista faríamos assim:



Uma lista tem índices. No caso acima, ela tem 4 posições, então os índices vão de 0 a 3. Note que o primeiro índice começa em 0, e isso é uma convenção. Por exemplo, o índice 0 representa o espaço com o nome “Davi”, já o índice 2 representa o espaço com o nome “Bela”.

Para manipularmos uma lista, podemos utilizar o nome da variável e, em seguida, o índice que desejamos acessar entre colchetes. Por exemplo, para escrevermos o que está no índice 2 da lista, fazemos assim:

```
print(variavelLista[2])
```

Abaixo, criamos uma lista e, em seguida, a imprimimos na tela. Após isso, mudamos o elemento de índice 2 e, por fim, imprimimos apenas esse novo elemento na tela:

```
lista=[1,2,3,4]
print(lista)
lista[2]=0
print(lista[2])
```

Podemos, também, por meio de funções, incluir, excluir e ver o número de elementos de uma lista. Para inserir um elemento ao final da lista, utilizamos a função append(), como a seguir:

```
lista=[1,2,3,4]
lista.append(5)
print(lista)
```

É possível, ainda, adicionarmos um elemento em qualquer posição de uma lista com a função insert(). Para utilizarmos essa função, colocamos, dentro dos parênteses, primeiro a posição onde desejamos inserir o elemento e depois o conteúdo dele. Como abaixo:

```
lista=[1,2,3,4]
lista.insert(0,5)
print(lista)
```

A função len() retorna o tamanho de uma lista, como no exemplo abaixo:

```
lista=[1,2,3,4]
print(len(lista))
```

Excluímos elementos com as funções del e clear. A função del exclui apenas um elemento, que vai entre parênteses; já a função clear exclui todos os elementos de uma lista. Notamos isso no próximo exemplo:

```
lista=[5,1,2,3,4]
del(lista[0])
print(lista)
lista.clear()
print(lista)
```

Também iremos estudar nesse capítulo as tuplas, que são um tipo de variável muito parecido com as listas. Porém, diferem principalmente delas por serem imutáveis. Na notação, a diferença é sútil. Ao invés de utilizarmos colchetes para representá-las, como fazemos com as listas, podemos utilizar parênteses ou não usar nenhum caracter delimitador, como se pode notar abaixo, na declaração de duas tuplas:

```
tupla1=1,2,3,4,5
tupla2=(6,7,8,9,10)
print(tupla1,tupla2)
```

Para último, temos os dicionários, que são listas nas quais podemos nos referir aos elementos por um nome preestabelecido, como no exemplo abaixo, que temos como intuito atrelar um endereço a uma pessoa:

```
dic={"gabriel":"casa 6","victor":"casa 2","davi":"casa 25”}
print(dic)
print(dic[“gabriel”])#imprimi na tela o endereco do gabriel
```

Adicionar um elemento ao final de um dicionário que já existe é bastante simples, basta fazer da seguinte forma:

```
dic={"gabriel":"casa 6","victor":"casa 2","davi":"casa 25”}
dic["isa"]="casa 1”
print(dic)
```

Podemos, através de funções,verificar tanto o tamanho quanto apagar elementos de um dicionário. A função del é utilizada para apagar um elemento. Por exemplo, se o Gabriel tivesse morrido, então iríamos querer excluí-lo do nosso dicionário, como no programa a seguir:

```
dic={"gabriel":"casa 6","victor":"casa 2","davi":"casa 25”}
print(dic)
del(dic["gabriel"])
print(dic)
```

A função len serve, como com as listas, para conferirmos o tamanho do dicionário. O código abaixo retrata isso:

```
dic={"gabriel":"casa 6","victor":"casa 2","davi":"casa 25”}
print(len(dic))
```

7.
ESTRUTURAS DE REPETIÇÃO

Neste capítulo, abordaremos os laços de repetição, também chamados de estruturas de repetição. Esse é um artifício utilizado nas linguagens de programação para a realização de uma tarefa repetitiva, a qual podemos, ou não, saber o número de vezes que será realizada. São basicamente 2 comandos de repetição em Python.

O primeiro comando é o for. Nessa estrutura, o bloco de código que está dentro dela será executado até que todos os elementos de um objeto iterável sejam percorridos. Um objeto iterável é, basicamente, um objeto que pode ser iterado, desmontado, como por exemplo: listas, strings, tuplas e dicionários.

O segundo é o comando while, que pode ser traduzido para o português como enquanto, ou seja, esse comando executa um bloco de códigos enquanto uma determinada condição for verdadeira.

```
//FOR

for variável in objeto iterável:
        comando(s)
```

```
//WHILE

while condição:
        comando(s)
```

Nos exemplos acima, temos as estruturas de ambos os comandos.

Notamos, ainda, a importância da função range, que é responsável por retornar uma série numérica, cujos limites são passados como parâmetros. Essa sequência pode ser

um objeto iterável dentro de um comando for. Como no exemplo abaixo, em que escrevemos os números de 1 a 100, utilizando tanto o comando for quanto o comando while:

```
for element in range(1,101):
        print(element)
```

```
i=1
while i<=100:
        print(i)
        i=i+1
```

A função range cria uma sequência de números e segue a seguinte sintaxe: range (início, final). Podemos notar que o primeiro número indica o início da sequência e o segundo, o final dela, sem incluí-lo. Ou seja, a série numérica começa no primeiro valor utilizado e termina no anterior ao último. No exemplo acima, o intervalo utilizado como parâmetro foi [1,101[. Sendo assim, vai do 1 ao 100.

8.
FUNÇÕES E BIBLIOTECAS

Neste capítulo, discutiremos funções e bibliotecas. Uma função é um conjunto de comandos que realiza uma tarefa específica em um módulo independente de código, como pode ser vista no exemplo a seguir. Vamos supor que nosso programa é uma empresa que faz bolos. Essa empresa disponibiliza para venda bolos simples, com recheio de chocolate ou com recheio de morango, e possui um gerente e um funcionário para fazer cada tipo. O gerente pede a um funcionário específico que faça um determinado sabor de bolo. Nesse exemplo, os funcionários que farão cada tipo do doce são as funções que assim serão chamadas; ou seja, sempre que desejarmos um de chocolate, designaremos a tarefa ao confeiteiro responsável por ele, que seguirá a receita dele e o entregará pronto.

Mas por que utilizamos funções? As funções nos dão diversas vantagens, dentre as quais destacaremos duas. A primeira delas é a economia de código, já que não precisamos reescrever tudo que está dentro da função toda vez que temos que usá-las, podendo apenas chamá-las. A segunda é que, por economizar linhas, o código fica mais simples e, portanto, mais legível. Um exemplo de função é uma função que escreve algum texto. Na prática, ela não tem utilidade, já que podemos realizar essa tarefa em apenas uma linha, mas é interessante para facilitar nosso entendimento.

```
def escreveTexto():
        print("texto")
escreveTexto()
```

No exemplo acima, criamos a função ”escreveTexto” e ela escreve "Texto” na tela. Depois de criarmos a função, chamamos, invocamo-las para que seja executada.
Para criarmos uma função, primeiro escrevemos o comando def e, em seguida, escrevemos o nome da função. Entre parênteses, escrevemos os parâmetros, que estudaremos logo em seguida. Outro exemplo, com um uso real mais provável, é uma função que verifica se um número é par ou ímpar e o escreve na tela. Daremos o nome dessa função de !parOuImpar!, como podemos notar a seguir:

```
num=int(input("escreva um numero\n”))
def parOuImpar(num):
        if (num%2)==0:
                print("este numero e par")
        else:
                print("este numero e impar")
parOuImpar(num)
```

Nesse exemplo, primeiro lemos um número do usuário com o comando input(). Em seguida, criamos a função parOuImpar, que escreve na tela se o número é par ou ímpar e, por fim, chamamos essa função.

É possível notar também que, logo após o nome da função, declaramos a variável num, que é chamada de parâmetro. Os parâmetros são os valores de entrada que uma função recebe. Através deles informamos valores que serão manipulados pela função. Esses, por sua vez, podem ser escritos tanto na forma de valores quanto de variáveis. No exemplo acima, o parâmetro é o número que a função deve analisar se é par ou ímpar.

Outro exemplo de função que podemos criar é a que efetua e escreve na tela o valor da soma de dois números.

```
def soma(num1,num2):
        print("a soma e {}".format(num1+num2))
soma(2,5)
```

Ainda é importante salientarmos a diferença entre variáveis locais e globais. As variáveis globais são declaradas fora de qualquer função e podem ser acessadas de qualquer lugar do programa. Já as variáveis locais são declaradas dentro de um escopo e, consequentemente, só podem ser acessadas dentro desse mesmo escopo. Um escopo é uma delimitação, em que valores e expressões estão associados.

```
def func1():
        num=0
print(num)
```

Um exemplo de variável local pode ser visto acima, no qual a variável num foi declarada dentro da func1, ou seja, o

programa acima apresentaria um erro, já que a variável num não está definida no escopo global. Outro exemplo é quando declaramos um parâmetro. Essa variável também só é válida dentro do escopo da função.

A seguir temos um exemplo de programa com retorno:

```
def func1():
        num1=int(input("digite dois numeros\n"))
        num2=int(input())
        return num1+num2
soma=func1()
print(soma)
```

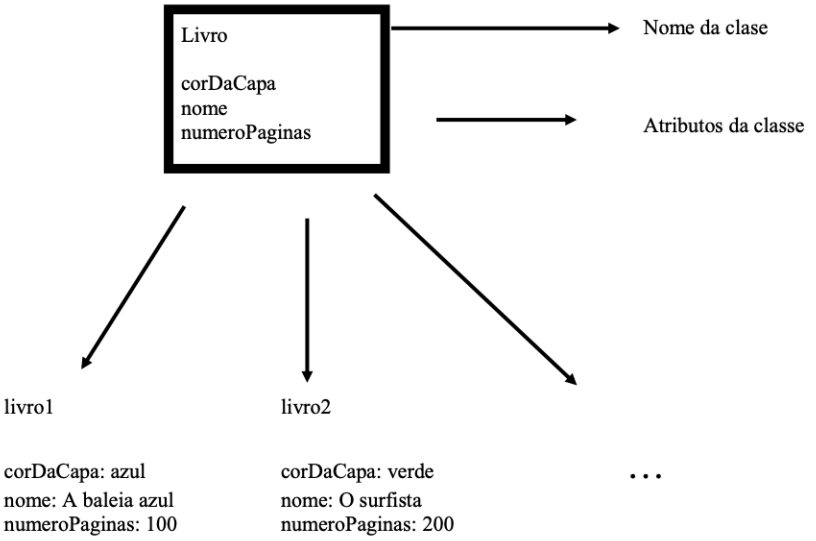
O retorno, como o próprio nome diz, retorna um valor. É importante recebermos em algum lugar o que foi retornado pela função. No exemplo acima, a função soma retorna para a variável soma. Uma função pode retornar mais de um valor e esses valores são separados por vírgulas.

Por fim, discutiremos as bibliotecas, mas de forma bem superficial. Uma biblioteca é, basicamente, um conjunto de funções pré-escritas, que podem ser utilizadas para solucionar certos problemas. Existem diversas bibliotecas para Python, dentre as quais destacam-se as que são usadas para a ciência de dados.

9.
FUNDAMENTOS DE POO

Para acabarmos os conteúdos referentes a Python, discutiremos o paradigma da programação orientada a objetos. A programação orientada a objetos (POO) é, como o nome diz, baseada em objetos, que são correspondentes aos da realidade, como: pessoa, carro, entre outros.

Nesse modelo existem classes, que são como um gabarito, um molde, para objetos com características semelhantes. Essas características na POO são chamadas, também, de atributos. Um exemplo de classe é a classe Livro, que origina os objetos livro1, livro2, etc. Na classe exemplificada abaixo, os atributos são: cor da capa, nome do livro e o número de páginas.



Os métodos são como funções, só que pertencentes às classes. Seria como se tivéssemos a classe pessoa e ela falasse. Para isso, criaríamos um método chamado falar. Sempre que desenvolvemos uma classe, a primeira coisa que devemos fazer é criar um construtor, que é um método "especial" que tem como objetivo criar os futuros objetos. Nota-se a estrutura de um construtor abaixo:

```
def __init__ (self, atributo1, atributo2, …):
        self.atributo1 = atributo1
        self.atributo2 = atributo2
        …
#usa-se dois underlines seguidos antes e depois da
#palavra init
```

Podemos observar que, antes de definirmos o construtor, utilizamos a palavra def, usada para declarar um método. Já a palavra init é reservada ao construtor. Abaixo, temos um exemplo real de uma classe pessoa que origina os objetos p1 e p2.

```
class Pessoa:
        def __init__(self, nome, idade):
                self.nome = nome
                self.idade = idade
        def falar(self):
                print('ola, tudo bem?')

p1 = Pessoa('Teo', 1)
p2 = Pessoa('Laura',10)
p1.falar()
```

No exemplo anterior, criamos, além de uma classe com atributos, o método falar, que imprime na tela a frase 'ola, tudo bem?'.

A palavra self serve para fazer referência à própria instância que está chamando o método e deve ser utilizada em todos os atributos que forem criados dentro de uma classe.

Temos ainda os métodos de classe, que são tipos “especiais” e se referem à classe e não ao objeto. Logo, não é requerida a criação de uma instância. Para implementarmos o método de classe, utilizamos o decorador @classmethod e, em seguida, criamos o método. Nesse tipo, ao invés de utilizamos a palavra self para nos referirmos à instância de uma classe, um objeto, utilizamos a palavra cls para fazer referência à classe em si. Abaixo, temos um exemplo, no qual o intuito é criarmos um objeto da classe pessoa. Ao invés de inserirmos o nome dela e sua idade, como no exemplo anterior, criaremos um objeto passando como parâmetro o nome e o ano de nascimento do indivíduo. Para isso, usaremos o método de classe.

```
class Pessoa:
        ano_atual = 2020
        def __init__(self, nome, idade):
                self.nome = nome
                self.idade = idade
        @classmethod
        def por_ano_nascimento(cls, nome, ano_nascimento):
                idade = cls.ano_atual - ano_nascimento
                return cls(nome, idade)
        def falar(self):
                print('ola, tudo bem?')
p1 = Pessoa.por_ano_nascimento('Teo', 2019)
```

Agora abordaremos os métodos estáticos, que são aqueles que não dependem nem da classe, nem da instância. Eles são como funções comuns, porém são colocados dentro de uma classe para facilitar a organização do código. Como por exemplo, se quisermos uma função que gera um id, ou seja, um número aleatório entre 1 e 1000, podemos torná-la em um método estático que faz parte da classe Pessoa.

```
from random import randint
class Pessoa:
        def __init__(self, nome, idade):
                self.nome = nome
                self.idade = idade
        @staticmethod
        def gera_id():
                rand = randint(1, 1000)
                return rand


print(Pessoa.gera_id())
#OU
p1 = Pessoa('Teo', 1)
print(p1.gera_id())
```

Notamos ainda que dentro do método estático gera_id(), utilizamos a função randint(), que gera números aleatórios.

Os atributos de classe são aqueles que se referem à classe e não aos objetos dela. Eles podem ser acessados pelos objetos, mas não por eles alterados. Observa-se isso no exemplo abaixo:

```
class A:
        atributoDeClasse=10

a1 = A()
a2 = A()
print(A.atributoDeClasse)
print(a2.atributoDeClasse)
print(a1.atributoDeClasse)
```

O Python, com relação ao encapsulamento, é diferente da maioria das outras linguagens de programação OO (orientadas a objeto). Nas outras, temos uma parte da classe privada ou protegida e uma parte pública. A parte privada ou protegida não pode ser acessada diretamente de fora do objeto, somente por getters e setters. Como em Python não existe esse tipo de divisão, foi convencionado que se um atributo ou método tiver um underline o precedendo, então é como se fosse privado. Porém, embora não seja recomendado, isso não nos impede de modificá-lo diretamente. Agora, caso esse atributo seja precedido por dois underlines, não poderemos acessá-lo diretamente, de modo algum. Para resolver esse problema, usamos a seguinte sintaxe:
nomeDoObjeto._nomeDaClass__nomeDoAtributo

Um exemplo disso é caso tivéssemos como intuito ter uma classe pessoa com o atributo nome “privado”, como pode ser visto abaixo:

```
class Pessoa:
        def __init__(self, nome, idade):
                self.__nome = nome
                self.idade = idade


p1 = Pessoa('Teo', 1)
print(p1._Pessoa__nome)
```

Podemos, também, tanto acessar quanto modificar esse atributo nome através de Getters e Setters. Os Getters servem para acessar um atributo “privado”; ou seja, são métodos que retornam um atributo. Já os Setters são métodos cuja função é receber um valor e, em seguida, a e ele igualar um atributo. Logo abaixo, temos o mesmo exemplo que utilizamos acima, mas resolvido de outra forma, com Geters e Setters:

```
class Teste:
        def __init__(self):
                self.__x = None
        @property
        def x(self):
                return self.__x
        @x.setter
        def x(self, value):
                self.__x = value
teste1 = Teste() #estamos criando o objeto teste1
teste1.x = 5 #setter
print(teste1.x) #getter
```

Podemos notar que, antes de definirmos os métodos Getter e Setter, utilizamos os decoradores @property para o get e @nomeDoAtributo.setter e, por fim, nas duas últimas linhas, invocamos primeiro o método set e, em seguida, o get.

Em um programa, podemos ter diversas classes e elas podem se relacionar entre si de 3 maneiras: associação, agregação e composição.

A associação é quando uma classe utiliza a outra, mas não existe uma relação de interdependência entre elas. Um exemplo disso é um programa em que uma classe escritor usa uma classe caneta, como pode ser visto a seguir:

```
class Escritor:
        def __init__(self, nome):
                self.nome = nome
                self.ferramenta = None
class Caneta:
        def __init__(self, marca):
                self.marca = marca
        def escrever(self):
                print('a caneta escreve')
escritor = Escritor(‘Teo')
caneta = Caneta('Bic')
escritor.ferramneta = caneta
escritor.ferramneta.escrever()
```

O segundo tipo é a agregação, na qual uma classe existe sem a outra, mas há uma relação de dependência entre elas. Como, por exemplo, uma classe carrinho de compras e uma

classe compra. Uma existe independente da outra, mas o carrinho de compras não faria sentido sem compras. Abaixo, temos a representação, em código, desse exemplo:

```
class CarrinhoDeCompras:
        def __init__(self):
                self.produtos = []
        def inserir_produto(self, produto):
                self.produtos.append(produto)
class Produto:
        def __init__(self, nome, valor):
                self.nome = nome
                self.valor = valor

p1 = Produto('caneta', 5)
carrinho = CarrinhoDeCompras()
carrinho.inserir_produto(p1)
```

Ainda temos a relação por composição, que é a mais forte. Nela, uma classe é “dona” da outra, isso quer dizer que, se a classe principal for apagada, todos os objetos que ela utilizou serão apagados também. Como, por exemplo, no código abaixo, no qual temos uma loja com inúmeras sedes e queremos armazenar os endereços em uma lista, utilizando uma classe endereço:

```
class class Loja:
        def __init__(self, nome):
                self.nome = nome
                self.enderencos = []
        def insere_enderecos (self, cidade, estado):
                self.enderencos.append(Endereco(cidade, estado))
class Endereco:
        def __init__(self, cidade, estado):
                self.cidade = cidade
                self.estado = estado


p1 = Loja('Renner')
p1.insere_enderecos('Curitiba', ‘PR')
p1.insere_enderecos('Guaratuba', 'PR')
print(p1.enderencos[0].cidade)
print(p1.enderencos[0].estado)
```

Pode-se notar no exemplo acima, que a classe loja contém todos os objetos endereços, ou seja, criaremos um novo objeto endereço sempre que invocarmos a classe Endereco.

Por fim, temos a herança, que serve, principalmente, para reaproveitarmos código. Vamos supor que temos uma classe pessoa que possui dois atributos, o atributo nome e o idade. Temos ainda uma classe estudante que, além do nome e da idade, possui o atributo ano escolar. Nesse caso, poderíamos herdar o nome e idade da classe pessoa, da seguinte forma:

```
class Pessoa:
        def __init__(self, nome, idade):
                self.nome=nome
                self.idade=idade
class Aluno(Pessoa):
        def __init__(self, nome, idade, anoEscolar):
                Pessoa.__init__(self, nome, idade)
                self.anoEscolar = anoEscolar

a1 = Aluno('Teo', 10, 'terceiro ano')
```

AUTORIA

Gabriel L. C. Selow tem 20 anos, mora em Curitiba, Paraná. É apaixonado por tecnologia e, atualmente, estuda Engenharia Eletrônica na UTFPR.