

# DESENVOLVIMENTO DE SOFTWARE

## PYTHON



**Link para exercícios em Python:**

<https://python.nilo.pro.br/exercicios3/>

# INTRODUÇÃO E HISTÓRIA DO PYTHON

- Python foi criado no final dos anos oitenta(1989) por Guido van Rossum no Centro de Matemática e Tecnologia da Informação (CWI, Centrum Wiskunde e Informatica), na Holanda.
- Python é uma linguagem de programação interpretada cuja filosofia enfatiza uma sintaxe favorecendo um código mais legível, além de ser “free”.

- É uma linguagem de programação multi-paradigma, pois suporta orientação de objeto, programação imperativa e, em menor escala, programação funcional.
- É uma linguagem interpretada, têm seus códigos fontes transformados em uma linguagem intermediária (específica de cada linguagem), que será interpretada pela máquina virtual da linguagem quando o programa for executado.
- Usa tipagem dinâmica e forte, isso, significa que o próprio interpretador do Python infere o tipo dos dados que uma variável recebe, sem a necessidade que o usuário da linguagem diga de que tipo determinada variável é.

# Aplicações do Python

Tem se tornado destaque em áreas atuais como: Machine Learning, Data Science, Big Data e Desenvolvimento Web.

- Para o **desenvolvimento web**, podemos contar com uma poderosa e simples linguagem, que possui dois importantes frameworks, o Django e o Flask. Com esses dois frameworks, podemos desenvolver aplicações web poderosas e que, com certeza, irão atender todas as demandas do mercado.

- O Python também é uma das principais tecnologias para trabalhar com Data Science. Muito disso por conta de sua simplicidade e bibliotecas para trabalhar com análise de dados, como o Pandas, uma das principais e mais poderosas do mercado.
- Big Data é a análise e interpretação de grandes volumes de dados. É, sem dúvidas, uma ferramenta fundamental para que as empresas possam obter vantagens competitivas em diversos segmentos. Assim como o Data Science e Machine Learning, o Python possui diversas bibliotecas para trabalhar com Big Data, já que estas áreas estão estritamente relacionadas. Várias são as bibliotecas para trabalhar com Big Data no Python, como a Pandas (citada anteriormente), NumPy, Matplotlib, Scikit-Learn, entre outras.

- Machine Learning, ou em sua tradução livre “Aprendizado de máquina”, é a área da ciência da computação que tem como objetivo a análise de dados que automatiza a construção de modelos analíticos. Assim como o Big Data, o Python é uma das principais tecnologias para trabalhar com Machine Learning, uma área de grande crescente na última década.

# Vantagens de utilizar o Python

- Aprender Python possui diversas vantagens, dentre elas podemos citar as seguintes:
  - Possui uma grande comunidade;
  - Multiplataforma;
  - Possui uma curva de aprendizagem baixa;
  - Pode ser utilizada em diversos segmentos;
  - É amplamente utilizado em diversas empresas, entre outras.



# Sintaxe do Python

- Conhecido por possuir uma sintaxe simples, o Python possui algumas características marcantes da linguagem:
  - Não utiliza ponto e vírgula (;) para finalizar uma instrução;
  - Utiliza indentação por espaços;
  - Uma variável pode armazenar diferentes tipos de dados;
  - Não há chaves ({} ) para delimitar o início e final de um bloco de código.

## Exemplo:

```
print('Meu primeiro programa em Python')

nome_variavel = 5

if nome_variavel == 5:
    print("O número é 5")
else:
    print("O número não é 5")
```

# BIBLIOTECAS E PACOTES

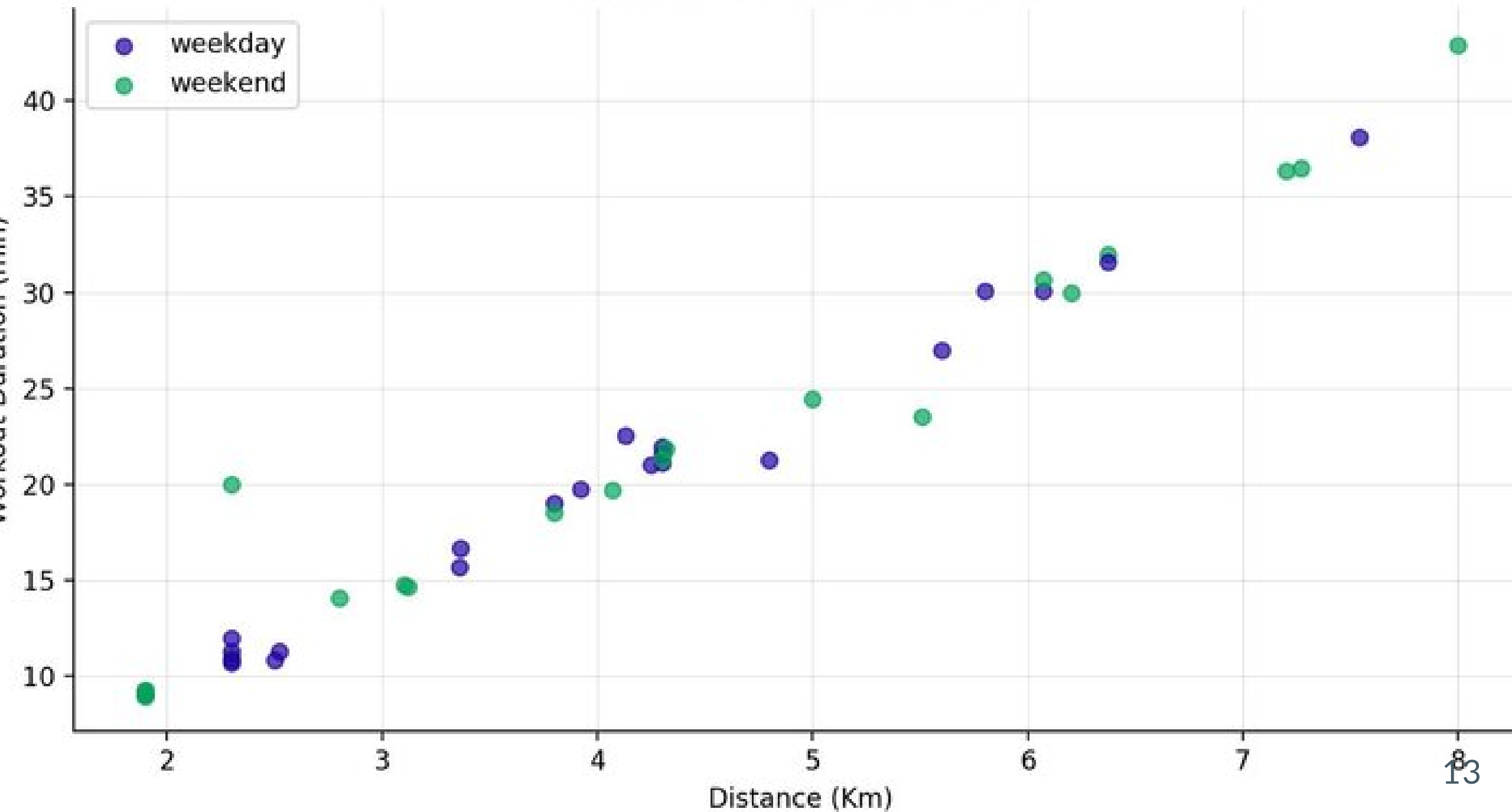
## Pillow

- Usando o Pillow, você pode não apenas abrir e salvar imagens, mas também influenciar o ambiente das imagens.
- O Pillow suporta muitos tipos de arquivos, como PDF, WebP, PCX, PNG, JPEG, GIF, PSD, WebP, PCX, GIF, IM, EPS, ICO, BMP e muitos outros.
- Com o Pillow, você pode criar facilmente miniaturas de imagens. As miniaturas carregam a maioria dos aspectos valiosos da sua imagem.

# Matplotlib

- O Matplotlib é uma biblioteca Python que usa o Python Script para escrever gráficos e plotagens bidimensionais. Frequentemente, aplicações matemáticas ou científicas exigem mais do que eixos únicos em uma representação. Essa biblioteca nos ajuda a criar várias ao mesmo tempo. No entanto, você pode usar o Matplotlib para manipular diferentes características das imagens.

Distance vs Workout Duration



# Numpy

- O Numpy é um pacote popular de processamento de array do Python. Ele fornece um bom suporte para diferentes objetos de matriz multidimensional. O Numpy não se limita apenas a fornecer matrizes, mas também fornece uma variedade de ferramentas para gerenciar essas matrizes. É rápido, eficiente e muito bom para gerenciar arrays e matrizes.

```
>>> a[(0,1,2,3,4), (1,2,3,4,5)]  
array([1, 12, 23, 34, 45])
```

```
>>> a[3:, [0,2,5]]  
array([[30, 32, 35],  
       [40, 42, 45],  
       [50, 52, 55]])
```

```
>>> mask = np.array([1,0,1,0,0,1], dtype=bool)  
>>> a[mask, 2]  
array([2, 22, 52])
```

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

# Utilizando o interpretador Python

- Você pode utilizar o interpretador no terminal do seu computador, digitando o comando:

```
python
```

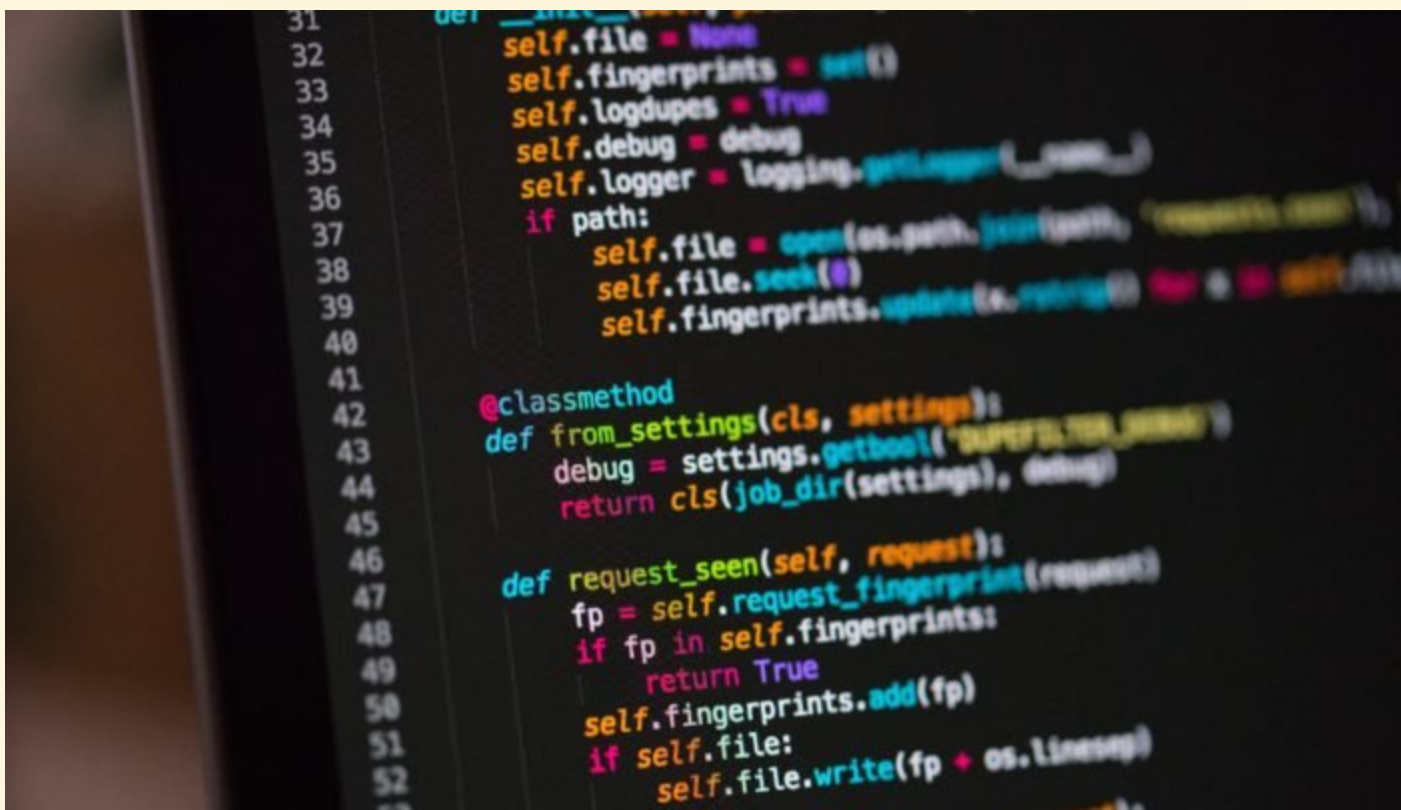
- Assim o python vai estar disponível no terminal.
- Para sair digite:

```
quit()
```



# A linguagem Python em si

- Agora vamos a sintaxe de uma das linguagens mais utilizadas do mercado, o Python.



```
31 def __init__(self, path):
32     self.file = None
33     self.fingerprints = set()
34     self.logdupes = True
35     self.debug = debug
36     self.logger = logging.getLogger(__name__)
37     if path:
38         self.file = open(os.path.join(path, 'requests.log'),
39                          'a')
40         self.file.seek(0)
41         self.fingerprints.update(self._get_fingerprints())
42
43 @classmethod
44 def from_settings(cls, settings):
45     debug = settings.getbool('SUPERSLOW_DEBUG')
46     return cls(job_dir(settings), debug)
47
48 def request_seen(self, request):
49     fp = self.request_fingerprint(request)
50     if fp in self.fingerprints:
51         return True
52     self.fingerprints.add(fp)
53     if self.file:
54         self.file.write(fp + os.linesep)
```

# VARIÁVEIS

- Variáveis são formas de se armazenar dados para uso.
- Variáveis são pequenos espaços de memória, utilizados para armazenar e manipular dados.
- As variáveis podem ser classificadas em 3 **tipos** básicos:
  - **int**: Número inteiro
  - **float**: Ponto flutuante (decimais)
  - **string**: Uma sequência de caracteres

- Ao contrário da maioria das outras linguagens, em Python, não é necessário declarar as variáveis que serão usadas, tampouco definir seu tipo. A própria sintaxe do dado a ser armazenado identifica o tipo da variável para armazená-lo.

### **Exemplo:**

Caso deseje-se atribuir o valor 3 à variável A, basta digitar `A = 3`. Python saberá que A é um inteiro (tipo “int”).

Por outro lado, se o valor a ser armazenado fosse 3,2 que é um dado do tipo “ponto flutuante”, este deveria ser expresso como `A = 3.2`.

Observe que, para Python, `A = 3 e B = 3.0` são variáveis de tipos diferentes e isto deve ser levado em conta ao se realizar certos tipos de manipulações de dados.

# SCRIPT DE AULA - Variáveis e Operadores

```
variavel = 5

variavel2 = '5'

variavel3 = 5.0

print(variavel3)

if variavel == variavel2:
    print('Isso é true')
else:
    print('Obviamente isso é false')
```

```
nome = input('Digite seu nome: ')\n\nprint(nome)
```

```
number = input('Digite um número: ')\n\nprint(number)\n\nprint(type(number))
```

```
number = int(input('Digite um número: '))\n\nprint(number)\n\nprint(type(number))
```

*# Declaração de Variáveis*

aula3 = 'Nome'

*# O Python não permite começar a declarar/definir uma variável com numerais*

3aula = 'Nome'

velocidade = 98

velocidade90 = 90

*# O Python não permite começar a declarar/definir uma variável com numerais*

90velocidade = 90

salario\_medio = 8000

*# O Python não permite espaço entre o nome da variável*

salario medio = 8000

\_salario = 5000

*# O Python não permite essa sintaxe dessa forma*

salario.medio = 8000

```
# Concatenação de Strings

print('Aula' + 'Python')

print('Aula ' + 'Python')

a = 'Aula '
b = 'de '
c = 'Python Deloitte'

# Uma quarta variável armazenando a minha concatenação

d = a + b + c

print(d)

# Concatenação direta

print(a + b + c + ' Concatenando direto')
```



```
# Manipulação de Strings

# len()

teste = 'Aula de Python'

print(len(teste))

# capitalize()

a = 'python'

b = 'python maravilha'

maiuscula = a.capitalize()
maiuscula2 = b.capitalize()

print(maiuscula)
print(maiuscula2)

# startswith()

c = 'Aula Python Deloitte'

d = c.startswith('Au')

print(d)

# upper()

nome = 'aula de python'

nomeUpper = nome.upper()

print(nome.upper())

print(nomeUpper)

# Fatiamento de Strings

s = 'Python'

# Seleciona os índices das posições 1 até 4, sem incluir o 4
print(s[1 : 4])

# Seleciona os índices da posição 2 até o fim
print(s[2: ])

# Seleciona os índices até a posição 3
print(s[ : 4])
```

```
# Faça uma rotina que receba uma string e depois transforme essa string em letras maiúsculas  
frase = input('Digite... ')  
fraseUpper = frase.upper()  
print(fraseUpper)
```

```
# Operações com Números
```

```
# Soma
```

```
soma = 5 + 5
```

```
print(soma)
```

```
print(type(soma))
```

```
# Subtração
```

```
subtracao = 5 - 5
```

```
print(subtracao)
```

```
# Multiplicação
```

```
multiplicacao = 5 * 5
```

```
print(multiplicacao)
```

```
# Divisão
```

```
divisao = 5 / 5
```

```
print(divisao)
```

```
print(type(divisao))
```

```
# Resto da Divisão
```

```
resto = 10 % 3
```

```
divisao = 10 / 3
```

```
print(resto)
```

```
print(divisao)
```

```
# Potência
```

```
potencia = 4 ** 2
```

```
print(potencia)
```

```
# Operadores de Comparação
```

```
# BOOLEANOS - BOOLEAN
```

```
# < >
```

```
menorQue = 5 < 10
```

```
print(menorQue)
```

```
maiorQue = 5 > 10
```

```
print(maiorQue)
```

```
# <= >= (MENOR OU IGUAL, MAIOR OU IGUAL)
```

```
menorIgual = 10 <= 10
```

```
print(menorIgual)
```

```
maiorIgual = 10 >= 10
```

```
print(maiorIgual)
```

```
# IGUAL ==
```

```
igual = 5 == 5
```

```
print(igual)
```

```
# Diferente
```

```
diferente = 5 != 5
```

```
print(diferente)
```

# OPERADORES LÓGICOS

<b>Operador</b>	<b>Descrição</b>	<b>Exemplo</b>
Not	NÃO	<b>not a</b>
And	E	<b>(a &lt;=10) and (c = 5)</b>
Or	OU	<b>(a &lt;=10) or (c = 5)</b>

# LISTAS - ARRAYS

- Lista é um conjunto sequencial de valores, onde cada valor é identificado através de um índice.
- O primeiro valor tem índice 0. Uma lista em Python é declarada da seguinte forma:

```
nome_lista = [valor1, valor2, valor3, ..., valorN]
```

- Uma lista pode ter valores de qualquer tipo, incluindo outras listas.

## Exemplo:

- Código:

```
L = [3 , 'abacate' , 9.7 , [5 , 6 , 3] , "Python" , (3 , 'j')]
```

```
print(L[2])  
# 9.7  
print(L[3])  
# [5, 6, 3]  
print(L[3][1])  
# 6
```

- Para alterar um elemento da lista, basta fazer uma atribuição de valor através do índice.
- O valor existente será substituído pelo novo valor.

## Exemplo:

```
L[3]= 'morango'  
print(L)  
# L = [3 , 'abacate' , 9.7 , 'morango' , "Python" , (3 , 'j')]
```



- A tentativa de acesso a um índice inexistente resultará em erro.

```
L[7]= 'banana'
```

```
Traceback (most recent call last):  
  File "<pyshell#4>", line 1, in <module>  
    L[7]='banana'  
IndexError: list assignment index out of range
```

# Funções para manipulação de arrays

- A lista é uma estrutura mutável, ou seja, ela pode ser modificada.

Exemplos a seguir:

Função	Descrição	Exemplo
<b>len</b>	retorna o tamanho da lista.	L = [1, 2, 3, 4] len(L) → 4
<b>min</b>	retorna o menor valor da lista.	L = [10, 40, 30, 20] min(L) → 10
<b>max</b>	retorna o maior valor da lista.	L = [10, 40, 30, 20] max(L) → 40
<b>sum</b>	retorna soma dos elementos da lista.	L = [10, 20, 30] sum(L) → 60
<b>append</b>	adiciona um novo valor na no final da lista.	L = [1, 2, 3] L.append(100) L → [1, 2, 3, 100]
<b>extend</b>	insere uma lista no final de outra lista.	L = [0, 1, 2] L.extend([3, 4, 5]) L → [0, 1, 2, 3, 4, 5]
<b>del</b>	remove um elemento da lista, dado seu índice.	L = [1,2,3,4] del L[1] L → [1, 3, 4]
<b>in</b>	verifica se um valor pertence à lista.	L = [1, 2 , 3, 4] 3 in L → True
<b>sort()</b>	ordena em ordem crescente	L = [3, 5, 2, 4, 1, 0] L.sort() L → [0, 1, 2, 3, 4, 5]
<b>reverse()</b>	inverte os elementos de uma lista.	L = [0, 1, 2, 3, 4, 5] L.reverse() L → [5, 4, 3, 2, 1, 0]

# Operações com arrays

- Concatenação ( + )

```
a = [0,1,2]
b = [3,4,5]
c = a + b
print(c)
# [0, 1, 2, 3, 4, 5]
```

- Repetição ( \* )

```
L = [1,2]  
R = L * 4  
print(R)  
# [1, 2, 1, 2, 1, 2, 1, 2]
```

# Fatiamento de arrays

- O fatiamento de listas é semelhante ao fatiamento de strings.

```
# seleciona os elementos das posições 1,2,3
```

```
L = [3 , 'abacate' , 9.7 , [5 , 6 , 3] , "Python" , (3 , 'j')]
```

```
L[1:4]
```

```
# ['abacate', 9.7, [5, 6, 3]]
```

```
# seleciona os elementos a partir da posição 2
```

```
L[2:]
```

```
# [9.7, [5, 6, 3], 'Python', (3, 'j')]
```

```
# seleciona os elementos até a posição 3
```

```
L[:4]
```

```
# [3, 'abacate', 9.7, [5, 6, 3]]
```

# Criação de arrays com range ( )

- A função range() define um intervalo de valores inteiros.
- Associada a list(), cria uma lista com os valores do intervalo.
- A função range() pode ter de 1 a 3 parâmetros:
  - range(n): gera um intervalo de 0 a n-1
  - range(i , n): gera um intervalo de i a n-1
  - range(i , n, p): gera um intervalo de i a n-1 com intervalo p entre os números



```
L1 = list(range(5))
```

```
print(L1)
```

```
# [0, 1, 2, 3, 4]
```

```
L2 = list(range(3,8))
```

```
print(L2)
```

```
# [3, 4, 5, 6, 7]
```

```
L3 = list(range(2,11,3))
```

```
print(L3)
```

```
# [2, 5, 8]
```

# TUPLAS

- Tupla, assim como a Lista, é um conjunto sequencial de valores, onde cada valor é identificado através de um índice.
- A principal diferença entre elas é que as tuplas são imutáveis, ou seja, seus elementos não podem ser alterados.
- Dentre as utilidades das tuplas, destacam-se as operações de empacotamento e desempacotamento de valores.
- Uma tupla em Python é declarada da seguinte forma:

```
nome_tupla = (valor1, valor2, ..., valorN)
```

## Exemplo:

```
T = (1,2,3,4,5)
print(T)
# (1, 2, 3, 4, 5)

print(T[3])
# 4
```

```
T[3] = 8
```

```
Traceback (most recent call last):  
  File "C:/Python34/teste.py", line 4, in <module>  
    T[3] = 8  
TypeError: 'tuple' object does not support item assignment
```

- Uma ferramenta muito utilizada em tuplas é o desempacotamento, que permite atribuir os elementos armazenados em uma tupla a diversas variáveis.

## Exemplo:

```
T = (10, 20, 30, 40, 50)
a, b, c, d, e = T

print("a = ", a, "b = ", b)
# a = 10 b = 20

print("d + e = ", d + e)
# d + e = 90
```

# DICIONÁRIOS

- Dicionário é um conjunto de valores, onde cada valor é associado a uma chave de acesso.
- Um dicionário em Python é declarado da seguinte forma:

```
Nome_dicionario = {  
    chave1 : valor1,  
    chave2 : valor2,  
    chave3 : valor3,  
    ...  
    chaveN : valorN  
}
```

## Exemplo:

```
D = { "arroz": 17.30, "feijão": 12.50, "carne": 23.90, "alface": 3.40 }  
print(D)  
# {'arroz': 17.3, 'carne': 23.9, 'alface': 3.4, 'feijão': 12.5}  
  
print(D["carne"])  
# 23.9  
  
print(D["tomate"])  
# ERRO KeyError: 'tomate'
```

- É possível acrescentar ou modificar valores no dicionário:

```
D["carne"] = 25.0
D["tomate"] = 8.80
print(D)
# { 'alface': 3.4 , 'tomate': 8.8, 'arroz': 17.3, 'carne': 25.0, 'feijão': 12.5 }
```

- Os valores do dicionário não possuem ordem, por isso a ordem de impressão dos valores não é sempre a mesma.



# Operações em dicionários

Comando	Descrição	Exemplo	
del	Exclui um item informando a chave.	<pre>del D["feijão"] print(D) {'alface':3.4, 'tomate':8.8, 'arroz':17.3, 'carne':25.0}</pre>	
in	Verificar se uma chave existe no dicionário.	<pre>"batata" in D False</pre>	<pre>"alface" in D True</pre>
keys()	Obtém as chaves de um dicionário.	<pre>D.keys() dict_keys(['alface', 'tomate', 'carne', 'arroz'])</pre>	
values()	Obtém os valores de um dicionário.	<pre>D.values() dict_values([3.4, 8.8, 25.0, 17.3])</pre>	

- Os dicionários podem ter valores de diferentes tipos.

## Exemplo:

```
Dx = {2: "carro", 3: [4, 5, 6], 7: ('a', 'b'), 4: 173.8}  
print(Dx[7])  
# ('a', 'b')
```

# **SCRIPT DE AULA - ARRAYS - DICIONÁRIOS**

```
# LISTAS - ARRAYS - São mutáveis
```

```
lista = [3, 45.6, ['Maria', 'João', 'Lucas'], 9, ('Alunos')]
```

```
lista2 = ['Mari', 'Ana', 'João', 'Lucas', 'Mateus']
```

```
print(lista)
```

```
print(lista[2])
```

```
print(lista[2][2])
```

```
# Alterar elemento do array
```

```
lista[1] = 50.1
```

```
print(lista)
```

```
# [3, 45.6, ['Maria', 'João', 'Lucas'], 9, 'Alunos']
```

```
# ['Maria', 'João', 'Lucas']
```

```
# Lucas
```

```
# [3, 50.1, ['Maria', 'João', 'Lucas'], 9, 'Alunos']
```

```

# OPERAÇÕES COM ARRAYS - LISTAS

A = [1, 2, 45, 67, 78, 100]

# len() - Retorna o TAMANHO do array (da lista). A quantidade de elementos e NÃO a posição deles

ALen = len(A)
print(ALen)

# min() - Retorna o menor valor da lista.

AMin = min(A)
print(AMin)

# sum() - Retorna a soma dos elementos da lista

ASum = sum(A)
print(ASum)

# append() - Adiciona um novo elemento no final do array (da lista)

A.append(200)
print(A)

# in - Verifica se um valor pertence ao array (lista)

B = ['Mari', 'Ana', 'João', 'Lucas', 'Mateus']

if ('Lucas' in B):
    print('True - Tem esse nome na lista')
else:
    print('False - Não existe esse nome na lista')

# Concatenação ( + ) e Repetição ( * )

d = [0, 1, 2, 3]
e = [4, 5, 6, 7, 8]

f = d + e
print(f)

g = d * 3
print(g)

# 6
# 1
# 293
# [1, 2, 45, 67, 78, 100, 200]
# True - Tem esse nome na lista
# [0, 1, 2, 3, 4, 5, 6, 7, 8]
# [0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3]

```

```
# TUPLAS - Elas são imutáveis
```

```
T = (32, 45, 67, 8899, 67)  
print(T)
```

```
print(T[3])
```

```
# T[3] = 'João' - Dá erro
```

```
# Desempacotamento
```

```
a, b, c, d, e = T
```

```
print(a, b, c, d, e)
```

```
# (32, 45, 67, 8899, 67)
```

```
# 8899
```

```
# 32 45 67 8899 67
```

```

# DICIONÁRIOS

D = {
    'arroz': 23.50, 'feijao': 13.90, 'alface': 11
}

print(D)
print(D['feijao'])

# in - Verifica se tal chave existe no dicionário

if ('verduras' in D):
    print('Existe sim')
else:
    print('Não existe')

# keys() - Retorna as chaves do dicionário

print(D.keys())

# values() - Retorna os valores do dicionário

print(D.values())

# Exercício Lanchonete

lanchonete = {
    'salgado': 4.50,
    'lanche': 6.50,
    'suco': 3.00,
    'refrigerante': 3.50,
    'doce': 1.00
}

print(lanchonete)

# {'arroz': 23.5, 'feijao': 13.9, 'alface': 11}
# 13.9
# Não existe
# dict_keys(['arroz', 'feijao', 'alface'])
# dict_values([23.5, 13.9, 11])
# {'salgado': 4.5, 'lanche': 6.5, 'suco': 3.0, 'refrigerante': 3.5, 'doce': 1.0}

```

# BIBLIOTECAS

- As bibliotecas armazenam funções pré-definidas, que podem ser utilizados em qualquer momento do programa.
- Em Python, muitas bibliotecas são instaladas por padrão junto com o programa.



- Para usar uma biblioteca, deve-se utilizar o comando import:

Exemplo: importar a biblioteca de funções matemáticas:

```
import math  
  
print(math.factorial(6))
```

Pode-se importar uma função específica da biblioteca:

```
from math import factorial  
print(factorial(6))
```

- Algumas bibliotecas padrão do Python:

Bibliotecas	Funções
math	Funções Matemáticas
tkinter	Interface Gráfica padrão
smtplib	e-mail
time	Funções de tempo

- Algumas bibliotecas externas para Python:

Bibliotecas	Funções
urllib	Leitor de RSS para uso na internet
numpy	Funções matemáticas mais avançadas
PIL/Pillow	Manipulação de imagens

# ESTRUTURAS DE DECISÃO

- As estruturas de decisão permitem alterar o curso do fluxo de execução de um programa, de acordo com o valor (Verdadeiro/Falso) de um teste lógico.
- Em Python temos as seguintes estruturas de decisão:

```
if (se)  
if..else (se...senão)  
if..elif..else (se...senão...senão se)
```

# Estrutura if

- O comando if é utilizado quando precisamos decidir se um trecho do programa deve ou não ser executado. Ele é associado a uma condição, e o trecho de código será executado se o valor da condição for verdadeiro.

Sintaxe:

```
if (condição):  
    Bloco de comandos
```

## Exemplo:

```
valor = int(input('Qual sua idade? '))  
  
if valor < 18:  
    print('Você ainda não pode dirigir.')
```

# Estrutura if ... else

- Nesta estrutura, um trecho de código será executado se a condição for verdadeira e outro se a condição for falsa.

Sintaxe:

```
if <condição>:  
    <Bloco de comandos para condição verdadeira>  
else:  
    <Bloco de comandos para condição falsa>
```

## Exemplo:

```
valor = int(input('Qual sua idade? '))

if (valor < 18):
    print('Você ainda não pode dirigir.')
else:
    print('Você é o cara! Pode dirigir.')
```



# Comando if ... elif ... else

- Se houver diversas condições, cada uma associada a um trecho de código, utiliza-se o elif.

Sintaxe:

```
if <condição1>:  
    <Bloco de comandos 1>  
elif <condição2>:  
    <Bloco de comandos 2>  
elif <condição3>:  
    <Bloco de comandos 3>  
    ::::::::::::::::::::::::::::::::::::::::::::  
else:  
    <Bloco de comandos default>
```

- Somente o bloco de comandos associado à primeira condição verdadeira encontrada será executado.
- Se nenhuma das condições tiver valor verdadeiro, executa o bloco de comandos default.

## Exemplo:

```
valor = int(input('Qual sua idade? '))

if valor < 6:
    print('Você é um bebê')
elif valor < 18:
    print('Você ainda não pode dirigir.')
elif valor > 60:
    print('Você está na melhor idade.')
else:
    print('Você é o cara! Pode dirigir')
```

# ESTRUTURAS DE REPETIÇÃO

- A Estrutura de repetição é utilizada para executar uma mesma sequência de comandos várias vezes.
- A repetição está associada ou a uma condição, que indica se deve continuar ou não a repetição, ou a uma sequência de valores, que determina quantas vezes a sequência deve ser repetida.
- As estruturas de repetição são conhecidas também como laços (loops).

# Laço while

- No laço while, o trecho de código da repetição está associado a uma condição.
- Enquanto a condição tiver valor verdadeiro, o trecho é executado.
- Quando a condição passa a ter valor falso, a repetição termina.

Sintaxe:

```
while <condição>:  
    <Bloco de comandos>
```

## Exemplo:

```
senha = '54321'
leitura = ''

while (leitura != senha):
    leitura = input('Digite a senha: ')
    if leitura == senha:
        print('Acesso liberado')
    else:
        print('Senha incorreta')
```

Exemplo: Encontrar a soma de 5 valores.

```
contador = 0
somador = 0

while contador < 5:
    contador = contador + 1
    valor = float(input('Digite o ' + str(contador) + ' valor: '))
    somador = somador + valor
print('Soma = ', somador)
```

# Laço for

- O laço for é a estrutura de repetição mais utilizada em Python.
- Pode ser utilizado com uma sequência numérica (gerada com o comando range) ou associado a uma lista.
- O trecho de código da repetição é executado para cada valor da sequência numérica ou da lista.



## Sintaxe:

```
for <variável> in range (início, limite, passo):  
    <Bloco de comandos >
```

*# ou*

```
for <variável> in <lista>:  
    <Bloco de comandos >
```

Exemplo: As notas de um aluno estão armazenadas em uma lista.  
Calcular a média dessas notas.

```
lista_notas = [3.4, 6.6, 8, 9, 8.8, 7.8]
soma = 0

for nota in lista_notas:
    soma = soma + nota
media = soma/len(lista_notas)
print('Média = ', media)
```

# FUNÇÕES

- Funções são pequenos trechos de código reutilizáveis. Elas permitem dar um nome a um bloco de comandos e executar esse bloco, a partir de qualquer lugar do programa.

# Como definir uma função

- Funções são definidas usando a palavra-chave `def`, conforme sintaxe a seguir:

```
def nome_função (definição dos parâmetros):  
    Bloco de comandos da função
```

**Obs.:** A definição dos parâmetros é opcional.

## Exemplo: Função simples

```
def hello():  
    print ("Olá Mundo!!!")
```

- Para usar a função, basta chamá-la pelo nome:

```
>>> hello()  
  
# Olá Mundo!!!
```

# Parâmetros e argumentos

- Parâmetros são as variáveis que podem ser incluídas nos parênteses das funções.
- Quando a função é chamada são passados valores para essas variáveis. Esses valores são chamados argumentos.
- O corpo da função pode utilizar essas variáveis, cujos valores podem modificar o comportamento da função.

## Exemplo: Função para imprimir o maior entre 2 valores

```
def maior(x, y):  
    if x > y:  
        print(x)  
    else:  
        print(y)
```

```
>>> maior(4, 7)
```

```
# 7
```

# Escopo das variáveis

- Toda variável utilizada dentro de uma função tem escopo local, isto é, ela não será acessível por outras funções ou pelo programa principal.
- Se houver variável com o mesmo nome fora da função, será uma outra variável, completamente independentes entre si.

Exemplo:



*# Função:*

```
def soma(x, y):  
    total = x + y  
    print("Total soma = ", total)
```

*# programa principal:*

```
total = 10  
soma(3, 5)  
print("Total principal = ", total)
```

- Resultado da execução:

```
# Total soma = 8  
# Total principal = 10
```

- Para uma variável ser compartilhada entre diversas funções e o programa principal, ela deve ser definida como variável global.
- Para isto, utiliza-se a instrução `global` para declarar a variável em todas as funções para as quais ela deva estar acessível. O mesmo vale para o programa principal.

Exemplo:

```
def soma(x,y):  
    global total  
    total = x+y  
    print("Total soma = ", total)  
  
# Programa principal  
  
global total  
total = 10  
soma(3, 5)  
print("Total principal = ", total)
```

- Resultado da execução:

```
# Total soma = 8  
# Total principal = 8
```

# Retorno de valores

- O comando `return` é usado para retornar um valor de uma função e encerrá-la.
- Caso não seja declarado um valor de retorno, a função retorna o valor `None` (que significa nada, sem valor).

## Exemplo:

```
def soma(x, y):  
    total = x + y  
    return total  
  
# programa principal  
  
s = soma(3, 5)  
print("soma = ", s)
```

- Resultado da execução:

```
# soma = 8
```

## Observações:

- O valor da variável total, calculado na função soma, retornou da função e foi atribuído à variável s.
- O comando após o return foi ignorado.

# Valor padrão

- É possível definir um valor padrão para os parâmetros da função. Neste caso, quando o valor é omitido na chamada da função, a variável assume o valor padrão.

Exemplo:

```
def calcula_juros(valor, taxa = 10):  
    juros = (valor * taxa)/100  
    return juros
```

```
>>> calcula_juros(500)  
# 50.0
```

# PROGRAMAÇÃO ORIENTADA A OBJETOS NO PYTHON:

- Vamos tratar sobre conceitos muito importante da Programação Orientada a Objetos.
  - Classes;
  - Objetos;
  - Herança;
  - Polimorfismo;
  - Encapsulamento.



## Conceito:

- A Programação Orientada a Objetos (POO) é um paradigma de programação baseado no conceito de Classes e Objetos.
- **Classes** podem conter dados e código:
  - Dados na forma de campos (também chamamos de atributos ou propriedades);
  - Código, na forma de procedimentos (frequentemente conhecido como métodos).

- Uma importante característica dos objetos é que seus próprios métodos podem acessar e frequentemente modificar seus campos de dados: objetos mantêm uma referência para si mesmo, o atributo `self` no Python.
- Na POO, os programas são projetados a partir de objetos que interagem uns com os outros.
- Esse paradigma se concentra nos objetos que os desenvolvedores desejam manipular, ao invés da lógica necessária para manipulá-los.

# Classes, Objetos, Métodos e Atributos

- As **Classes** são tipos de dados definidos pelo desenvolvedor que atuam como um modelo para objetos. Pra não esquecer mais: Classes são fôrmas de bolo e bolos são objetos.
- **Objetos** são instâncias de uma Classe. Objetos podem modelar entidades do mundo real (Carro, Pessoa, Usuário) ou entidades abstratas (Temperatura, Umidade, Medição, Configuração).

- **Métodos** são funções definidas dentro de uma classe que descreve os comportamentos de um objeto. Em Python, o primeiro parâmetro dos métodos é sempre uma referência ao próprio objeto.
- Os **Atributos** são definidos na Classe e representam o estado de um objeto. Os objetos terão dados armazenados nos campos de atributos. Também existe o conceito de atributos de classe, mas veremos isso mais pra frente.

# Princípios Básicos de POO

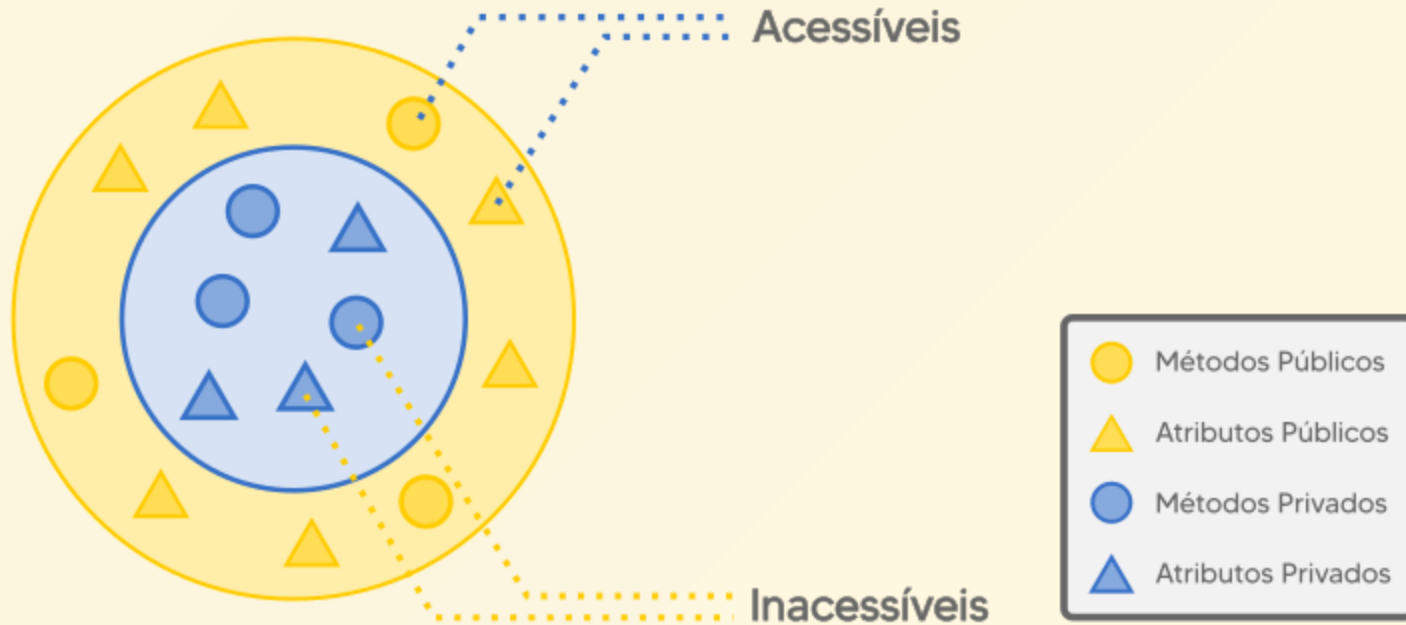
- A programação orientada a objetos é baseada nos seguintes princípios:

## Encapsulamento

- Usamos esse princípio para juntar, ou encapsular, dados e comportamentos relacionados em entidades únicas, que chamamos de objetos.
- Por exemplo, se quisermos modelar uma entidade do mundo real, por exemplo Computador.

- Encapsular é **agregar** todos os atributos e comportamentos referentes à essa Entidade dentro de sua Classe.
- Dessa forma, o mundo exterior não precisa saber como um Computador liga e desliga, ou como ele realiza cálculos matemáticos.
- Basta **instanciar** um objeto da Classe Computador, e utilizá-lo.
- O princípio do Encapsulamento também afirma que informações importantes devem ser contidas dentro do objeto de maneira **privada** e apenas **informações selecionadas** devem ser **expostas** publicamente.

- Veja a imagem abaixo que exemplifica a relação entre atributos e métodos públicos e privados:



- A implementação e o estado de cada objeto são mantidos de forma privada dentro da definição da Classe.
- Outros objetos não têm acesso a esta classe ou autoridade para fazer alterações.
- Eles só podem chamar uma lista de funções ou métodos públicos.
- Essa característica de ocultação de dados fornece maior segurança ao programa e evita corrupção de dados não intencional.



# Abstração

- Pense em um Tocador de DVD.



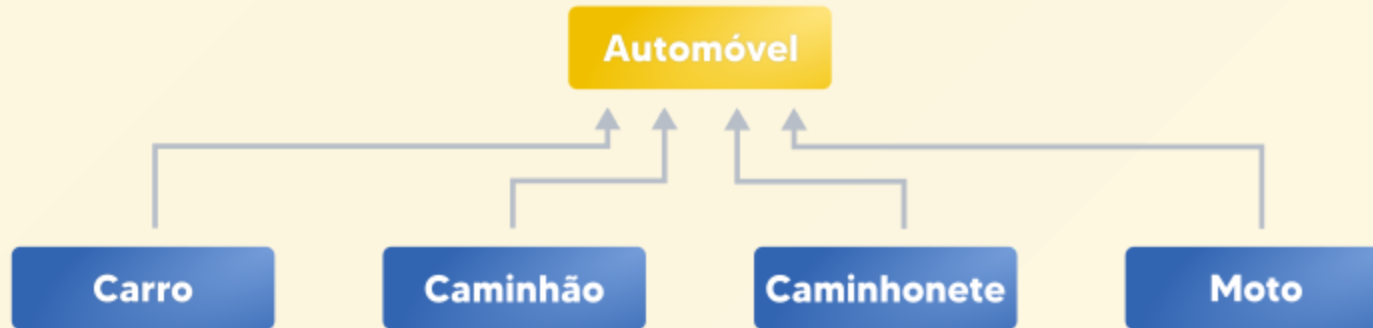
- Ele tem uma placa lógica bastante complexa com diversos circuitos, transistores, capacitores e etc do lado de dentro e apenas alguns botões do lado de fora.
- Você apenas clica no botão de “Play” e não se importa com o que acontece lá dentro: o tocador apenas... Toca.
- Ou seja, a complexidade foi “escondida” de você: isto é Abstração na prática.
- O Tocador de DVD abstraiu toda a lógica de como tocar o DVD, expondo apenas botões de controle para o usuário.

- O mesmo se aplica à Classes e Objetos: **nós podemos esconder atributos e métodos do mundo exterior**. E isso nos traz alguns benefícios:
  - Primeiro, a interface para utilização desses objetos é muito mais simples, basta saber quais “botões” utilizar.
  - Também reduz o que chamamos de “Impacto da mudança”, isto é: ao se alterar as propriedades internas da classes, nada será alterado no mundo exterior, já que a interface já foi definida e deve ser respeitada.

# Herança

- Herança é a característica da POO que possibilita a reutilização de código comum em uma relação de **hierarquia entre Classes**.
- Vamos utilizar a entidade **Carro** como exemplo.
- Agora imagine uma **Caminhonete**, um **Caminhão** e uma **Moto**.
- Todos eles são **Automóveis**, correto? Todos possuem **característica semelhantes**, não é mesmo?
- Podemos pensar que Automóveis **aceleram**, **freiam**, possuem mecanismo de **acionamento** de **faróis**, entre outros.

- Uma relação de hierarquia entre as classes poderia ser pensada da seguinte forma:



- Dessa forma podemos modelar os comportamentos semelhantes em uma **Classe “pai”: Automóvel** que conterà os atributos e comportamentos comuns.
- Através da Herança, as **Classes filhas de Automóvel vão herdar** esses atributos e comportamentos, sem precisar reescrevê-los reduzindo assim o tempo de desenvolvimento.

# Polimorfismo

- Quando utilizamos Herança, teremos Classes filhas utilizando código comum da Classe acima, ou **Classe pai**.
- Ou seja, as Classes vão compartilhar **atributos** e **comportamentos** (herdados da Classe acima).
- Assim, Objetos de Classes diferentes, terão métodos e atributos compartilhados que podem ter implementações diferentes, ou seja, um método pode possuir várias formas e atributos podem adquirir valores diferentes.

- Daí o nome: Poli (muitas) morfismo (formas).
- Para entendermos melhor, vamos utilizar novamente o exemplo da entidade **Carro** que herda de **Automóvel**.
- Suponha agora que **Automóvel** possua a definição do método `acelerar()`.
- Por conta do conceito de Polimorfismo, objetos da **Classe Moto** terão uma implementação do método `acelerar()` que será diferente da implementação desse métodos em instâncias da Classe Carro.



# Programação Orientada a Objetos no Python

Agora vamos finalmente juntar a Programação Orientada a Objetos com o Python.

Python possui palavras reservadas (**keywords**) para criarmos Classes e Objetos:

👉 Primeiro, temos a keyword `class` que utilizamos para criar uma classe.

👉 Também temos a keyword `self`, utilizada para guardar a referência ao próprio objeto.

💡 Uma observação importante, caso você venha de outra linguagem de programação: Python não utiliza a keyword `new` para instanciar novos objetos.

- Exemplo prático:
  - Vamos criar uma classe que representa uma entidade do tipo Pessoa.

- Ela deve ter os seguintes campos:
  - Nome como String;
  - Idade como Inteiro;
  - Altura como Decimal.
- Também deve ter métodos para:
  - Dizer “Olá”;
  - Cozinhar;
  - Andar.

► Utilizando POO e Python, podemos modelar a entidade Pessoa da seguinte forma:

```
class Pessoa:
    def __init__(self, nome: str, idade: int, altura: float):
        self.nome = nome
        self.idade = idade
        self.altura = altura

    def dizer_ola(self):
        print(f'Olá, meu nome é {self.nome}. Tenho {self.idade} '
              f'anos e minha altura é {self.altura}m.')

    def cozinhar(self, receita: str):
        print(f'Estou cozinhando um(a): {receita}')
```

```
    def andar(self, distancia: float):
        print(f'Saí para andar. Volto quando completar {distancia} metros')
```

## 👉 Código explicado:

- Temos a definição da **Classe** na primeira linha com `class Pessoa`. Isso diz ao Python que vamos criar a definição de uma nova classe.
- Em seguida, temos o método `__init__`. Ele é muito importante e é chamado de **Construtor**. Ele é chamado ao se **instanciar** objetos e é nele que geralmente setamos os atributos do objeto.
- Em seguida temos a definição dos métodos `dizer_ola()`, `cozinhar()` e `andar()`.
- Perceba que no método `dizer_ola()` referenciamos os atributos do próprio objeto com o argumento `self: self.nome`, `self.idade` e `self.altura`.

👉 Agora veja como podemos instanciar e interagir com objetos dessa Classe:

```
# Instancia um objeto da Classe "Pessoa"
pessoa = Pessoa(nome = 'João', idade = 25, altura = 1.88)

# Chama os métodos de "Pessoa"
pessoa.dizer_ola()
pessoa.cozinhar('Spaghetti')
pessoa.andar(750.5)

# Resultados:

# Olá, meu nome é João. Tenho 25 anos e minha altura é 1.88m.
# Estou cozinhando um(a): Spaghetti
# Saí para andar. Volto quando completar 750.5 metros
```

💡 Se lembra do Construtor?

👉 Ele entrou em ação na linha 2 do código acima.

👉 Quando escrevemos `pessoa = Pessoa()`, chamamos o método `__init__` da classe `Pessoa`, passando os parâmetros `nome`, `idade` e `altura`.

# PADRÃO MVC - ARQUITETURA MVC

👉 O **MVC** é uma sigla do termo em inglês **Model** (modelo) **View** (visão) e **Controller** (Controle) que facilita a troca de informações entre a interface do usuário aos dados no banco, fazendo com que as respostas sejam mais rápidas e dinâmicas.

- Você já parou para pensar o que se passa por trás de uma tela de login de um software?
- Em frações de segundos a página é capaz de absorver as informações que foram digitadas no campo de email e senha, realizar a validação e entregar uma resposta positiva ou negativa.



- Esse processo só se torna possível quando existe um padrão de arquitetura de software adequado. Embora exista vários que podem ser utilizados, o MVC é o mais conhecido e empregado entre os desenvolvedores profissionais.

## O que é MVC?

- Apesar de muitas pessoas considerarem essa sigla como um padrão de design de interface, **na verdade ele é um padrão de arquitetura de software responsável por contribuir na otimização da velocidade entre as requisições feitas pelo comando dos usuários.**

- Com quase 50 anos de formulação, a arquitetura **MVC** é dividida em três componentes essenciais: `Model`, `Controller` e `View`.
- Uma dúvida muito recorrente na programação é se no processo de desenvolvimento pode ter apenas esses 3 componentes ou se é possível acrescentar mais alguns.
- A resposta é sim para a possibilidade de inserir uma camada extra. Essa sequência de códigos pode ser alterada conforme a necessidade do projeto.
- **Mas um código com muitas camadas se torna muito confuso e por isso, o ideal é manter o padrão original.**

# Model ou Modelo

- **Sua responsabilidade é gerenciar e controlar a forma como os dados se comportam por meio das funções, lógica e regras de negócios estabelecidas.**
- Ele é o detentor dos dados que recebe as informações do Controller, válida se ela está correta ou não e envia a resposta mais adequada.

# Controller ou Controlador

- **A camada de controle é responsável por intermediar as requisições enviadas pelo View com as respostas fornecidas pelo Model**, processando os dados que o usuário informou e repassando para outras camadas.
- Numa analogia bem simplista, o controller operaria como o 'maestro de uma orquestra' que permite a comunicação entre o detentor dos dados e a pessoa com vários questionamentos no MVC.

# View ou Visão

- **Essa camada é responsável por apresentar as informações de forma visual ao usuário.** Em seu desenvolvimento devem ser aplicados apenas recursos ligados a aparência como mensagens, botões ou telas.
- Seguindo nosso processo de comparação o View está na linha de frente da comunicação com usuário e é responsável transmitir questionamentos ao controller e entregar as respostas obtidas ao usuário. É a parte da interface que se comunica, disponibilizando e capturando todas as informação do usuário.

# Como os componentes interagem?

- Tudo começa com a interação do usuário na camada View. A partir daí o controlador pega essas informações e envia para o Model que fica responsável por avaliar aqueles dados e transmitir uma resposta.
- O controlador recebe essas respostas e envia uma notificação de validação daquela informação para a camada visão, fazendo com a mesma apresente o resultado de maneira gráfica e visual.

# Conclusão

👉 O MVC funciona como um padrão de arquitetura de software que melhora a conexão entre as camadas de dados, lógica de negócio e interação com usuário. Através da sua divisão em três componentes, o processo de programação se torna algo mais simples e dinâmico.