

Nivelamento Python



Renato Rodrigues Oliveira da Silva
renato.oliveira@ifsp.edu.br





Sumário

- Tipos e Estruturas de Dados
- Condicionais e Estruturas de Repetição
- Funções
- Tratamento de Erros
- Orientação a Objetos

Tipos de Dados

- Numéricos
 - Inteiros: 1, 2, 3, 4, ...
 - Ponto Flutuante: 1.5, 2.7, 1.1, ...
- Texto (String)
 - Iniciam com aspas simples ou duplas
 - 'String', "Outro string", ...
 - Caracteres especiais: \n, \t, \\\, \', \", etc.
- Outros: Fractional, Complex, ...

Tipos de Dados

- Conversão entre os tipos
 - Função **float(valor)**: float(1), float('1.0'), ...
 - Função **int(valor)**: int(1.0), int('1'), ...
 - Função **str(valor)**: str(2.5), str(10), ...
- Variáveis criadas no momento da atribuição =
- Tipo inferido dinamicamente
 - a = 1 # int
 - b = 1.5 # float
 - c = 'Teste' # string
 - c[0] == 'T', c[0:2] == 'Te', c[-1] == 'e'

Tipos de Dados

- Operações algébricas

- Números

- Soma (+): $3 + 2$ #5
 - Subtração (-): $5 - 1$ #4
 - Multiplicação (*): $4 * 3$ #12
 - Divisão (/): $3/2$ #1.5
 - Divisão Inteira (//): $3/2$ #1
 - Módulo (%): $3 \% 2$ #0.5
 - Exponenciação (**): $3 ** 2$ #9

- Strings

- Concatenação (+): 'a' + 'b' # 'ab'
 - 'a' + 3 # Erro! Primeiro converter 3 em str
 - Multiplicação (*): 'a' * 5 # 'aaaaa'



Entrada / Saída

- Entrada de dados: **input('Mensagem')**
 - Exibe a mensagem no console e espera a os dados de entrada do usuário. Retorna os dados digitados após pressionada a tecla ENTER.
 - O valor retornado é sempre um string.
- Saída de dados: **print (valor)**
- Exemplo

```
nome = input ('Digite o seu nome: ')
idade = input ('Digite a sua idade: ')
print ('O usuário ' + nome + ' tem ' + str(idade) + 'anos')
```



Funções do tipo String

- Algumas funções do tipo String
 - **upper**: converte em letras maiúsculas
 - Ex: `'abc'.upper()` # `'ABC'`
 - **lower**: converte em letras minúsculas
 - `'ABC'.lower()` # `'abc'`
 - **split(sep)**: cria uma lista de palavras, delimitadas por sep
 - `'Um dois três'.split()` # `['Um', 'dois', 'três']`
 - **join(iterable)**: cria um novo string, a partir da concatenação da coleção passada como parâmetro
 - `str.join(['A', 'B', 'C'])` # `'ABC'`

Funções tipo String

- **format(args):** Cria um novo string, contendo os argumentos passados como parâmetro.
 - Os caracteres {} são substituídos no string pelo valor dos argumentos, em sequência.
 - Ex: 'Um - {}, Dois - {}'.format(1,2) # Um - 1, Dois - 2
- Mais informações:
 - <https://docs.python.org/2/library/stdtypes.html#string-methods>



Tipos de Dados

Exercício 1

Pedir ao usuário para informar o valor de três valores numéricos e calcular a média

Tipos de Dados

Exercício 1 - Solução

```
valor1 = float(input('Digite o primeiro valor:'))  
valor2 = float(input('Digite o segundo valor:'))  
valor3 = float(input('Digite o terceiro valor:'))  
media = (valor1 + valor2 + valor3) / 3  
print('A média é {}'.format(media))
```



Estruturas de Dados

- O Python implementa estruturas de dados, utilizadas para agrupar/acessar valores
- Exemplos de estruturas de dados
 - Listas
 - Tuplas
 - Sets
 - Dicionários

Listas

- Tipo mutável, definido por colchetes, com valores separados por vírgulas
 - lista1 = []
 - lista2 = ['um', 'dois', 'três']
 - lista3 = ['A', 1.0, 1]
- Elementos podem ser acessados especificando a posição na lista
 - lista2[1] == 'dois'
 - lista2[-1] == 'três'
 - lista2[1:3] == ['dois', 'três'] #retorna uma nova lista



Listas

- Elementos podem ser adicionados utilizando a função **append**, e removidos com **remove**
 - `lista = ['a']`
 - `lista.append('b')` # `lista == ['a', 'b']`
 - `lista.remove('a')` # `lista == ['b']`
- Também é possível utilizar operações algébricas em listas
 - `lista = [1, 2]`
 - `lista * 2` # `[1, 2, 1, 2]`
 - `lista + [3, 4]` # `[1, 2, 3, 4]`

Listas

- A função **len** permite saber o tamanho da lista
 - `len([1, 2, 3, 4]) == 4`
- Listas também podem conter outras listas
 - `matriz3x3 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]`
 - `matriz3x3[0] == [1,2,3]`
 - `matriz3x3[0][0] = 1`
- Para verificar se um elemento pertence a uma lista, utilizamos o operador **in**
 - `lista1 = [1, 2, 3, 4, 5]`
 - `1 in lista1` **#True**
 - `6 in lista1` **#False**

Tuplas

- Conjunto **imutável** de elementos, definidos por parênteses, com valores separados por vírgulas
 - `tupla1 = ('a', 'palavra', 1.2, 10)`
 - `tupla1[1] == 'palavra'`
 - `tupla2 = (1,)`
- Acesso mais rápido em relação a tipos mutáveis
- Tuplas podem possuir tipos mutáveis
 - `tupla2 = (1.7, ['a', 'b'])`
 - `tupla2[1].append('c') # tupla2 == (1.7, ['a', 'b', 'c'])`



Sets

- Conjunto não ordenado de elementos que não permite duplicatas de elementos
- É definido por chaves, com elementos separados por vírgulas
 - `set1 = {'a', 'b', 'c', 'c'}`
 - `set1 == {'a', 'b', 'c'} # True, remove a duplicata`
 - `set2 = set('abracadabra')`
 - `set2 == {'a', 'r', 'b', 'c', 'd'} # letras únicas`



Sets

- O uso envolve a remoção de duplicatas, teste de pertinência ao grupo e operações matemáticas com outros grupos (união $|$, intersecção $\&$, etc.)
- Exemplos
 - $\text{set1} = \{1, 2, 3\}$
 - $\text{set2} = \{3, 4, 5\}$
 - $\text{set3} = \text{set1} \& \text{set2} == \{3\}$ # Intersecção
 - $\text{set4} = \text{set1} | \text{set2} == \{1, 2, 3, 4, 5\}$ # União
 - $\text{set5} = \text{set1} - \text{set2} == \{1, 2\}$ # Diferença



Exercício 2

- Considere a lista de funcionários de uma empresa
 - engenheiros → [João, Nice, Sueli, Amanda]
 - programadores → [João, Sofia, Rafael, Amanda]
 - gerentes → [Nice, Sueli, Rafael, João]
- Usando Sets, imprima o nome de todos os funcionários da empresa
- Crie e imprima o set dos funcionários que são engenheiros e programadores
- Crie e imprima o set dos programadores e gerentes

Exercício 2 - Solução

```
engenheiros = {'João', 'Nice', 'Sueli', 'Amanda'}  
programadores = {'João', 'Sofia', 'Rafael', 'Amanda'}  
gerentes = {'Nice', 'Sueli', 'Rafael', 'João'}  
todos = engenheiros | programadores | gerentes  
print('Todos os funcionarios: {}'.format(todos))  
e_p = engenheiros & programadores  
print('Engenheiros e programadores: {}'.format(e_p))  
p_g = programadores & gerentes  
print('Programadores e gerentes: {}'.format(p_g))
```

- Permite criar conjuntos de associações entre tipos (chave → valor)
- As chaves devem ser tipos imutáveis (como strings, números ou tuplas)
 - `dict1 = {1 : 'Um', 2 : 'Dois', 3 : 'Tres'}`
 - `dict1[1] == 'Um' # acesso ao elemento pela chave`
 - `dict1[1] = 'Quatro' # substituição`
`#dict1 == {1 : 'Quatro', 2 : 'Dois', 3 : 'Tres' }`

- Para adicionar elementos, criamos uma nova associação
 - `dict1[10] = 'Dez'`
 - `dict1 == {1 : 'Quatro', 2 : 'Dois', 3 : 'Tres', 10 : 'Dez'}`
- Para remover elementos, utilizamos **pop(chave)**
 - `dict1.pop(1)`
 - `dict1 == {2 : 'Dois', 3 : 'Tres', 10 : 'Dez'}`

Exercício 3

- Escreva um programa que leia do usuário o nome e RA de 3 alunos e armazene essa informação em um dicionário, relacionando o RA ao nome do aluno
- Peça ao usuário para informar um RA e exiba o nome do aluno associado

Exercício 3 – Solução

```
nome1 = input('Informe o nome do primeiro aluno:')
ra1 = input('Informe o RA do primeiro aluno:')
nome2 = input('Informe o nome do segundo aluno:')
ra2 = input('Informe o RA do segundo aluno:')
nome3 = input('Informe o nome do terceiro aluno:')
ra3 = input('Informe o RA do terceiro aluno:')
alunos = {ra1:nome1, ra2:nome2, ra3:nome3}
ra_pesquisa = input('Informe um RA para pesquisar:')
print('O aluno com esse RA é {}'.format(alunos[ra_pesquisa]))
```

Condicionais

- Permitem controlar o fluxo de execução do programa por meio de expressões lógicas

if condição:

comando1

comando2

...

else:

comando3

...

- A indentação define os blocos de execução

Condicionais

- **Exemplo**

```
idade = int( input ('Digite a sua idade:') )
```

```
if idade > 18:
```

```
    print('Maior de idade')
```

```
else:
```

```
    print('Menor de idade')
```

Condicionais

- O comando `elif` permite verificar várias condições, e executar um fluxo diferente para cada uma

```
if condição1:
```

```
...
```

```
elif condição2:
```

```
...
```

```
elif condição3:
```

```
....
```

```
else:
```

```
...
```



Condicionais

- **Exemplo**

```
altura = float( input('Digite a sua altura:') )
```

```
if altura <= 1.50:
```

```
    print('Baixo')
```

```
elif 1.50 < altura <= 1.80:
```

```
    print('Médio')
```

```
else:
```

```
    print('Alto')
```

Condicionais

- Para executar um mesmo fluxo para diversas condições, podemos utilizar os operadores **and**, **not** e **or**

if (condição1 **and** condição2) **or** condição3:

...

elif condição2 **and** (**not** condição4):

...

else:

...

Condicionais

Exercício 4

- Escreva um programa para calcular uma folha de pagamento simplificada, sabendo que os descontos do Imposto de Renda dependem do salário bruto
 - Até 1.903,98 → isento
 - Entre 1.903,99 até 2.826,65 → desconto de 7,5%
 - Entre 2.826,66 até 3.751,05 → desconto de 15%
 - Entre 3.751,06 até 4.664,68 → desconto de 22,5%
 - Acima de 4.664,68 → desconto de 27,5%
- Imprima o valor do salário líquido na tela

Condicionais

Exercício 4 – Solução

```
salario = float(input('Informe o salario bruto:'))
desconto = 0
if 1903.99 < salario <= 2826.65:
    desconto = salario * 1.075
elif 2826.65 < salario <= 3751.05:
    desconto = salario * 1.15
elif 3751.06 < salario <= 4664.68:
    desconto = salario * 1.225
elif 4664.68 < salario:
    desconto = salario * 1.275
print('O salario liquido é {}'.format(salario - desconto))
```



Operador Ternário

- É possível utilizar condicionais como o operador ternário da linguagem C (?).

a if c else b

- A expressão retorna o valor **a** se a condição **c** for verdadeira. Se for false, retorna o valor **b**
- Exemplo
 - situacao_aluno = 'aprovado' if nota > 10 else 'reprovado'
 - paciente = 'doente' if temperatura > 36 else 'sadio'

Operadores and e or

- Diferentemente de muitas linguagens, os operadores and e or não retornam valores booleanos, mas sim os valores comparados
- Operador **and**
 - 'a' and 'b' #retorna b
 - 'a' and 'b' and 'c' #retorna c
 - " and 'a' and 'b' #retorna "
 - Os valores são avaliados da esquerda para a direita, e retorna o primeiro valor 'false', caso exista. Caso contrário o último valor 'true'
- Valores considerados 'false': 0, "", (), [], {} e None

Operadores and e or

- Operador **or**
 - 'a' or 'b' # retorna a
 - " or 'b' # retorna b
 - " or () or {} #retorna {}
 - Os valores são avaliados da esquerda para a direita, e retorna o primeiro valor 'true', caso exista. Caso contrário o último valor 'false'
- Exemplos de aplicação
 - a = 'primeiro'
 - b = 'segundo'
 - 0 and a or b # retorna 'segundo'
 - 1 and a or b # retorna 'primeiro'

Operadores and e or

Exercício 5

- Leia do 3 valores das notas de um aluno e calcule a média
- Armazene a média em uma variável, e utilize o operador ternario “if else” para informar a situação do aluno
 - Média < 5 → Reprovado
 - Média ≥ 5 → Aprovado

Operadores and e or

Exercício 5 – Solução

```
nota1 = float(input('Informe a primeira nota:'))  
nota2 = float(input('Informe a segunda nota:'))  
nota3 = float(input('Insira a terceira nota:'))  
media = (nota1+nota2+nota3)/3  
print('Aprovado' if media >= 5 else 'Reprovado')
```

Técnicas de Repetição

- for
- while
- break, continue, else
função range
- list comprehensions
- reversed, sorted

Comando for

- Itera sobre os itens de uma sequência (lista, tupla, caracteres de um string, etc.)
- Para cada item da sequência, os comandos do bloco definido pelo for são executados

for item **in** sequencia:

 comando1

 comando2

- Exemplo

```
lista = [1, 2, 3, 4, 5]
```

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

Comando while

- Executa as instruções dentro do bloco definido no while enquanto a condição for avaliada como verdadeira

while condição:

comando1

comando2

...

Comando while

- Exemplo: Imprimir os elementos de uma lista em sequência

```
lista = ['a', 'b', 'c', 'd', 'e']
```

```
indice = 0
```

```
while indice < len(lista):
```

```
    print(lista[indice])
```

```
    indice += 1
```


continue, pass e break

- O comando **pass** não realiza nenhuma operação
 - Pode ser usado quando um comando é requerido sintaticamente, mas nenhuma ação é necessária
- O comando **continue** interrompe a iteração de uma estrutura de repetição, passando para a próxima iteração da sequência
- O comando **break** interrompe a execução da estrutura de repetição

continue, pass e break

- Exemplo: Imprimir os numeros pares de uma lista

```
lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
for item in lista:
```

```
    if item % 2: # não é divisível por 2? continua...
```

```
        continue
```

```
    print(item)
```

continue, pass e break

- Exemplo: Verificar se existe um numero maior que 100 na lista

```
lista = [1, 20, 32, 19, 53, 41, 77, 800, 91, 18]
```

```
achou = False
```

```
for item in lista:
```

```
    if item > 100:
```

```
        achou = True
```

```
        break
```

```
    # finaliza a execução do for
```

Função Range

- Permite criar uma progressão numérica de valores
- `range(fim)`
 - cria uma sequência iniciando em 0, até “fim”
- `range(inicio, fim, inc)`
 - cria uma sequência iniciando em “inicio”, até “fim”, com “inc” incrementos a cada passo
- Exemplo: Imprime sequência de 0 a 99
for **i in** `range(100)`:
 `print(i)`

Técnicas de Repetição

Exercício 6

- Desenvolva um gerador de tabuada, capaz de gerar a tabuada de qualquer número inteiro entre 1 a 10. O usuário deve informar de qual numero ele deseja ver a tabuada. Exemplo:

Tabuada de 5:

$$5 \times 1 = 5$$

$$5 \times 2 = 10$$

...

$$5 \times 10 = 50$$



Técnicas de Repetição

Exercício 6 – Solução

```
numero = int(input('Informe o numero para a tabuada:'))  
for n in range(1,11):  
    print('{} x {} = {}'.format(numero, n, numero*n))
```





Funções

- Definição
- Documentação
- Passagem de argumentos: posição, keywords
- Passagem de parâmetros: `*args`, `**kwargs`
- Parâmetros padrão

Funções

- Definem um bloco de código reutilizável, com comportamento e responsabilidades bem definidas.
- Podem retornar um resultado, e utilizar parâmetros como entrada

```
def nome_da_função(parâmetros):  
    comando1  
    comando2  
    ...  
    return valor
```


Funções

- É possível adicionar um string para detalhar informações sobre a função
- O string deve ser adicionado logo após a definição do nome da função, e deve usar 3 aspas (simples ou duplas)

```
def nome_da_função(parâmetros):
```

```
    “ Essa função calcula o ... ”
```

```
    comando1
```

```
    comando2
```

```
    ...
```

```
    return valor
```



Funções

- **Exemplo:** Definição da função

```
def area_retangulo(base, altura)  
    “Calcula a área de um retangulo”  
    return base * altura
```

- **Exemplo:** Uso da função

```
valor_area = area_retangulo(5, 10)  
print('O valor da área é ' + str(valor_area))
```



Parâmetros com valores padrão

```
def confirma(msg, tentativas = 3, alerta = 'Tente novamente!'):
    while True:
        resposta = input(msg)
        if resposta in ('s', 'si', 'sim'):
            return True
        elif resposta in ('n', 'no', 'não'):
            return False
        tentativas = tentativas - 1
        if tentativas < 0:
            raise ValueError('Resposta inválida')
    print(alerta)
```

Parâmetros com valores padrão

```
def confirma(msg, tentativas = 3, alerta = 'Tente novamente!'):
```

- Possíveis chamadas à função:

Por posição

- confirma('Sair do programa?')
- confirma('Substituir o arquivo?', 2)
- confirma('Substituir o arquivo?', 2, 'Escolha sim ou não')

Por palavra-chave

- confirma('Substituir o arquivo?', alerta = 'Sim ou não')
- confirma('Substituir o arquivo?', alerta = 'S/n', tentativas = 2)

Número variável de parâmetros

- Um parâmetro ***args** guarda quaisquer argumentos passados para a função além da definição formal
 - Os valores são guardados em uma **tupla**

```
def soma(primeiro, segundo, *outros):  
    resultado = primeiro + segundo  
    for elemento in outros:  
        resultado = resultado + elemento  
soma(2, 3, 6, 8, 10)
```

Número variável de parâmetros

- Um parâmetro ****kwargs** guarda os argumentos passados por palavra-chave, além da definição formal
 - Os valores são guardados em um **dicionário**

```
def imprimir_receita(item, quantidade, **outros):  
    print ('Item - {}, Quantidade - {}'.format(item, quantidade))  
    for elemento in outros:  
        print ('Item - {}, Quantidade - {}'.format(elemento, outros[elemento]))  
imprimir_receita('sal', '1 colher', queijo = '50g', tomate = '1')
```

Desempacotar argumentos

- Se os valores a serem passados para a função já estiverem em uma lista, tupla ou dicionário, podemos “desempacota-los” diretamente
 - Por **posição**:

```
# passagem por posição
```

```
list(range(3,6)) # [3, 4, 5]
```

```
# ou desempacotando uma lista
```

```
arg = [3,6]
```

```
list(range(*arg)) # [3, 4, 5]
```

Desempacotar argumentos

- Por **palavra-chave**:

```
def area_retangulo(base, altura):
```

```
    ...
```

```
# passagem por palavra-chave
```

```
area_retangulo(base = 10, altura = 5)
```

```
# ou desempacotando um dicionário
```

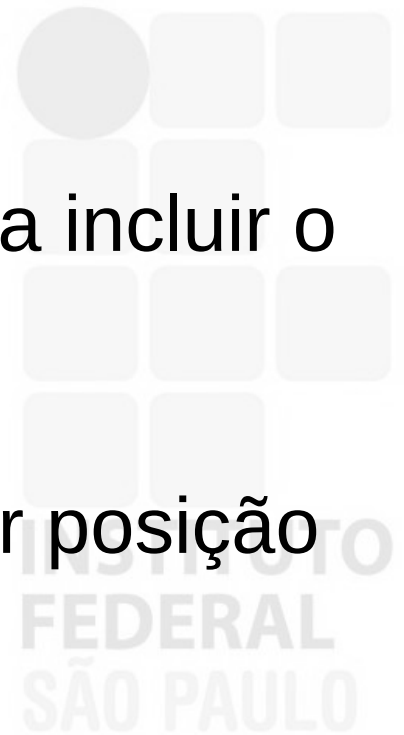
```
arg = { base : 10, altura : 5 }
```

```
area_retangulo(**arg)
```


Funções

Exercício 8

- Escreva a função `soma_imposto` com dois parâmetros
 - taxa: quantia de imposto sobre vendas expressa em porcentagem
 - custo: valor de um item antes do imposto
- A função deve retornar o valor de custo para incluir o imposto sobre vendas.
- Escreva um programa que use a função `somaImposto`, passando os argumentos por posição e por palavra-chave



Exercício 8 – Solução

```
def soma_imposto(custo, taxa):
```

```
    return custo * taxa
```

```
valor_bruto = float(input('Informe o valor bruto:'))
```

```
taxa_imposto = float(input('Informe a aliquota do imposto:'))
```

```
# chamada por posicao
```

```
valor_liquido = soma_imposto(valor_bruto, taxa_imposto)
```

```
# chamada por palavra-chave
```

```
valor_liquido = soma_imposto(custo = valor_bruto,  
                             taxa = taxa_imposto)
```

Tratamento de Erros

- Erros que acontecem durante a execução do programa são chamados ***Exceptions***
- *Exceptions* podem ser tratados para evitar que o programa trave
- O tratamento de *exceptions* é um modo conveniente de tratamento de erros de diferentes naturezas, de modo flexível

Tratamento de Erros

- Sintaxe

```
try:
```

```
...
```

```
except tipo_da_exception1:
```

```
... # tratamento da exception1
```

```
except tipo_da_exception2:
```

```
... # tratamento da exception2
```

Tratamento de Erros

- **Exemplo:** Cálculo do Índice de Massa Corpóreo

try:

```
altura = float(input('Digite a altura (metros):')
```

```
peso = float(input('Digite o peso:')
```

```
imc = peso / (altura ** 2)
```

except ZeroDivisionError:

```
print('Peso não pode ser zero!')
```

except ValueError:

```
print('Digite apenas números!')
```

Tratamento de Erros

Exercício 14

- Altere o exercício 3, que relaciona nomes de alunos (valores) e RAs (chaves) para tratar o caso de consultar uma chave inexistente no dicionário
 - *Exception* KeyError



Tratamento de Erros

Exercício 14 – Solução

try:

....


```
alunos = {ra1:nome1, ra2:nome2, ra3:nome3}
```

```
ra_pesquisa = input('Informe um RA para pesquisar:')
```

```
print('O aluno com esse RA é {}'.format(alunos[ra_pesquisa]))
```

except KeyError:

```
print('RA inexistente!')
```



Orientação a Objetos

- Classes
- Atributos, Métodos
- Herança



Orientação a Objetos

- Orientação a objetos permite modelar um projeto utilizando um paradigma de entidades (objetos) que possuem **características** e **comportamentos**, promovendo:
 - Abstrair a complexidade do código, ocultando detalhes do desenvolvedor (objetos podem ser utilizados como uma caixa preta)
 - Reuso de código (Herança)
 - Tratamento de erros (Exceções)

Classes e Métodos

- Classes definem um “molde” para criação de objetos
- Nas classes podem ser definidos os atributos (variáveis – características) e métodos (funções – comportamentos)
- A definição dos métodos é similar à definição de uma função: **def** nome_do_método
- Métodos e atributos são acessados pelo operador .
 - objeto.atributo

Classes e Métodos

- Exemplo 1

```
class Carro:
```

```
    “Exemplo de classe”
```

```
    def __init__(self): # inicializa um novo objeto
```

```
        self.velocidade = 0
```

```
    def acelerar(self, velocidade):
```

```
        self.velocidade = self.velocidade + velocidade
```

```
carro = Carro() # cria um objeto tipo Carro
```

```
carro.acelerar(100) # altera a velocidade do objeto
```



Classes e Métodos

- Exemplo 2

```
class Carro:
```

```
    “Exemplo de classe”
```

```
    def __init__(self, velocidade): # inicializa um novo objeto
```

```
        self.velocidade = velocidade
```

```
    def acelerar(self, velocidade):
```

```
        self.velocidade = self.velocidade + velocidade
```

```
carro = Carro(40) # cria um objeto tipo Carro
```

```
carro.acelerar(60) # altera a velocidade do objeto
```

Variáveis de Classe e Instância

- Variáveis de classe são definidas no corpo da classe, e os valores são compartilhados por todos os objetos
- Variáveis de instância são definidas dentro do método `__init__` e os valores são únicos para cada objeto

Variáveis de Classe e Instância

```
class Cachorro:
```

```
    truques = [] # variável de classe
```

```
    def __init__(self, nome):
```

```
        self.nome = nome
```

```
    def add_truque(self, truque):
```

```
        self.truques.append(truque)
```

```
a = cachorro('Rex')
```

```
a.add_truque('rolar')
```

```
b = cachorro('Pluto')
```

```
b.add_truque('fingir morto')
```

truques == ['rolar', 'fingir morto']

Variáveis de Classe e Instância

```
class Cachorro:
```

```
    def __init__(self, nome):
```

```
        self.nome = nome
```

```
        truques = [] # variável de instância
```

```
    def add_truque(self, truque):
```

```
        self.truques.append(truque)
```

```
a = cachorro('Rex')
```

```
a.add_truque('rolar')
```

```
b = cachorro('Pluto')
```

```
b.add_truque('fingir morto')
```

truques == ['rolar']

truques == ['fingir morto']

Herança

- Herança permite que uma classe aproveite (herde) as características e comportamentos de classes pai
 - Diminuindo a redundância no desenvolvimento de projetos
- Definição: `class ClasseFilha(ClassePai)`
- A classe filha pode então estender o comportamento da classe pai, sobrescrevendo métodos e adicionando novos métodos e atributos

Herança

- Exemplo:

```
class Pessoa:
    def __init__(self, nome):
        self.nome = nome
    def __str__(self):
        return self.nome
```

```
p = Pessoa('Fulano')
print(p) # Fulano
```


```
class Aluno(Pessoa):
    def __init__(self, nome, ra):
        super().__init__(nome)
        self.ra = ra
    def __str__(self):
        return self.nome + ' ' + ra
```

```
a = Aluno('Ciclano', '1234')
print(a) # Ciclano 1234
```

Orientação a Objetos

Exercício 13

- Altere o módulo `area.py` para criar 3 classes: Objeto, Retangulo e Circulo
 - A classe Objeto deve possuir o método vazio `calcula_area()`
 - As classes Retangulo e Circulo devem herdar de Objeto e alterar o método `calcula_area()` para computar os valores correspondentes



Orientação a Objetos

Exercício 13 – Solução

```
class Objeto:  
    def calcula_area(self):  
        pass
```



Orientação a Objetos

Exercício 13 – Solução

```
class Retangulo(Objeto):  
    def __init__(self, base, altura):  
        self.base = base  
        self.altura = altura  
    def calcula_area(self):  
        return base * altura
```

Orientação a Objetos

Exercício 13 – Solução

```
class Circulo(Objeto):  
    def __init__(self, raio):  
        self.raio = raio  
    def calcula_area(self):  
        return 3.1415 * (raio**2)
```



Referências

- <https://docs.python.org/3/tutorial/>
- <http://www.pythonforbeginners.com/error-handling/exception-handling-in-python>
- <https://pythontips.com/2013/08/04/args-and-kwargs-in-python-explained/>
- <https://realpython.com/blog/python/primer-on-python-decorators/>
- <https://pythonspot.com/en/objects-and-classes/>



Referências

- <https://www.python-course.eu/lambda.php>
- <https://regexone.com/references/python>