INSTITUTO DO EMPREGO E FORMAÇÃO PROFISSIONAL

PESSOAS 2030 | PORTUGAL 2030 | Cofinanciado pela União Europeia

Os Fundos Europeus mais próximos de si.

REPÚBLICA PORTUGUESA | TRABALHO, SOLIDARIEDADE E SEGURANÇA SOCIAL

Programação avançada com Python

POO

Objetos do mundo real vs da programação

Objetos do mundo real

- No mundo real, tal como na Programação Orientada a Objetos (POO), os objetos caracterizam-se por terem atributos e comportamentos:

Bateria

Atributos:

Tamanho

Tensão

Capacidade

(...)

Comportamentos:

Fornecer energia

Indicar nível de bateria

(...)

Carro

Atributos:

Cor

Marca

Nível do depósito

(...)

Comportamentos:

Acelerar

Acender as luzes

Abrir os vidros

(...)

Objetos do mundo real

- Da mesma forma que acontece no mundo real, os comportamentos dos objetos podem influenciar ou ser influenciados pelos seus atributos:

Comportamento: acelerar

```
self.nivel_deposito -= 1
```

i Ao acelerar, o depósito vai perdendo gradualmente "1" de quantidade.

Atributo: Nível do depósito


```
self.nivel_deposito = 100
```


```
self.nivel_deposito = 50
```


```
self.nivel_deposito = 0
```

Objetos da programação

- Para a programação, tudo o que for constituído por atributos e comportamentos poderá ser visto como um objeto. Desta forma, na POO, podemos dizer que um médico ou um cão poderão ser vistos como objetos.

Médico

Atributos:	Comportamentos:
Nome	Consultar
Experiência	Diagnosticar
Especialidade	Aconselhar
(...)	(...)

Cão

Atributos:	Comportamentos:
Cor	Ladrar
Perigoso	Morder
Idade	Correr
(...)	(...)

Objetos da programação

- Para que possamos entender melhor este conceito, imagine: cada vez que um médico consulta ou faz um diagnóstico, a sua experiência cresce.

Comportamento:
Consultar / Diagnosticar

```
self.experiencia += 1
```

Atributo:
Experiência

! O comportamento consultar ou diagnosticar, faz com que o atributo experiência possa evoluir.

Objetos da programação

Exercício 1

No bloco de notas ou folha de papel, identifique separadamente **atributos** e **comportamentos** para os seguintes objetos (pelo menos 5 de cada).

Se possível, indique ainda alguns atributos que possam influenciar os seus respetivos comportamentos.

Telemóvel

Professor

Objetos da programação

Exercício 1 (solução)

Telemóvel

Atributos:

- Marca
- Tamanho
- Nível da bateria
- Total de chamadas
- Total de mensagens
- Aplicativos instalados
- Contactos
- Saldo

Comportamentos:

- Atender chamadas
- Enviar SMS
- Receber SMS
- Adicionar contactos
- Remover contactos
- Aceder à internet
- Instalar aplicativos
- Remover aplicativos

Professor

Atributos:

- Nome
- Data de nascimento
- Morada
- NIF
- Email
- Salário
- Contratado
- Experiência
- Grupo de ensino

Comportamentos:

- Ensinar
- Ouvir
- Tirar dúvidas
- Fazer testes
- Fazer fichas
- Preparar a aula
- Avaliar
- Fazer sumários
- Marcar faltas

POO

Classes

Classes

- As classes tratam-se de uma descrição pormenorizada de atributos (designados por propriedades) e comportamentos (designados por métodos) que cada objeto, destas resultante, terá.
- Imagine a classe como um código genético que dará mais tarde origem aos objetos de si resultantes.

Classe

```
class Ovelha:
    def __init__(self, nome):
        self.nome = nome

    def falar(self):
        print(f"{self.nome} diz Mehhhh")

    def comer(self, alimento):
        print (f"{self.nome} a comer {alimento}")

    def olhar(self, algo):
        print(f"{self.nome} a olhar {algo}")
```

Objeto

! O Objeto reúne todas as características e funcionalidades da classe.

```
ovelha_dolly = Ovelha("Dolly")
```

Classes

- A mesma classe (ou seja, o mesmo “código genético”) pode dar origem a objetos distintos, ainda assim, todos os seus objetos terão atributos (propriedades) e comportamentos (métodos) comuns.

Classe

```
class Ovelha:
    def __init__(self, nome):
        self.nome = nome

    def falar(self):
        print(f"{self.nome} diz Mehhhh")

    def comer(self, alimento):
        print (f"{self.nome} a comer {alimento}")

    def olhar(self, algo):
        print(f"{self.nome} a olhar {algo}")
```

Objetos resultantes da classe:

```
ovelha_sherk = Ovelha("Sherk")
ovelha_chris = Ovelha("Chris")
ovelha_luna = Ovelha("Luna")
```

Objetos a realizar os comportamentos para o qual foram programados:


```
ovelha_sherk.olhar("câmara")
ovelha_chris.comer("erva")
ovelha_luna.comer("erva")
```

```
Sherk a olhar câmara
Chris a comer erva
Luna a comer erva
```

Classes

De cada classe (“código genético”) resultam obrigatoriamente sempre objetos do mesmo tipo. Tal como, por exemplo, do código genético de uma ovelha, só podem resultar ovelhas e nunca gatos ou cães:

Classes: conceitos importantes

Na programação orientada a objetos (POO), existem os seguintes conceitos fundamentais que devem ser compreendidos com claresa: classe, instanciar, instância e objeto e variável objeto.


```
class Ovelha:
    def __init__(self, nome):
        self.nome = nome

    def falar(self):
        print("Mehhhh!")
```


```
ovelha_dolly = Ovelha("Dolly")
```

! Cada variável objeto representa o seu objeto (ou instância).

Objeto ou instância:

```
Ovelha("Dolly")
```

Variável objeto:

```
ovelha_dolly
```

! Apesar de não o ser, muitas vezes a variável objeto é referida como sendo o objeto concreto, uma vez que o representa.

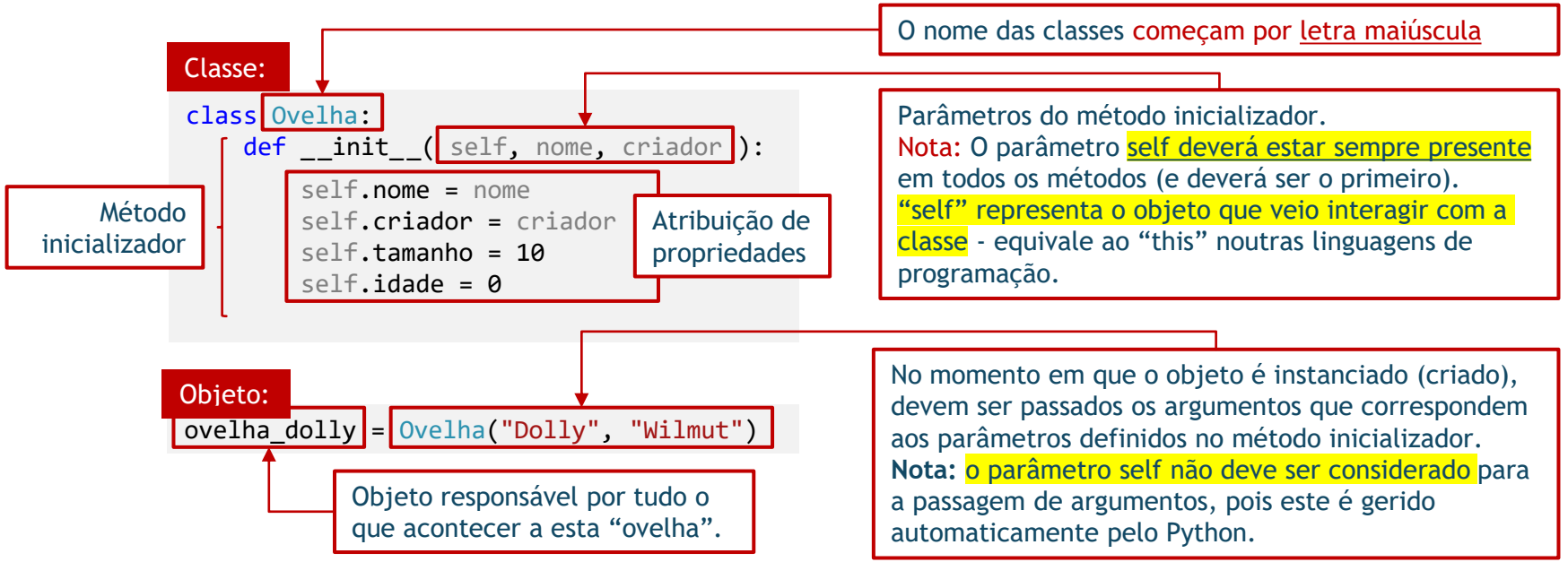
! Embora os termos 'objeto' e 'instância' sejam frequentemente usados como sinónimos, não significam necessariamente o mesmo. Dependendo da linguagem, é possível criar objetos sem classes (exemplo: objetos literais), pelo que, nesses casos, não devem ser considerados instâncias. O termo 'instância' é utilizado para indicar que um determinado objeto foi criado a partir de uma classe.

POO

Métodos

Método inicializador

- O método inicializador (“__init__” ➛ com 2 underscores de cada lado) é o primeiro método a ser executado aquando da instanciação (criação) de um objeto. A sua principal função é atribuir aos objetos os seus respetivos atributos (propriedades) e realizar outros processos adjacentes que se vejam necessários no momento da criação de cada objeto.
- **Curiosidade:** Em outras linguagens de programação este método é apelidado de método construtor.



Método inicializador

Exercício 1

Crie uma classe que possa representar um pirilampo.

Sabe-se que o pirilampo poderá nascer inicialmente com:

- **Nome:** configurável
- **Idade:** zero
- **Tamanho:** zero
- **Cor:** configurável
- **Intensidade do brilho:** zero

Crie ainda dois pirilampos provenientes da sua classe:

- “Luciferino” com a cor verde
- “Vivido” com a cor amarelo

- Cábula:

```
class Ovelha:
    def __init__(self, nome, criador):
        self.nome = nome
        self.criador = criador
        self.tamanho = 10
        self.idade = 0
```

```
ovelha_dolly = Ovelha("Dolly", "Wilmut")
```

Método inicializador

Exercício 1 (solução)

Classe

Objetos

```
class Pirilampo:
    def __init__(self, nome, cor):
        self.nome = nome
        self.idade = 0
        self.tamanho = 0
        self.cor = cor
        self.intensidade_brilho = 0

pirilampo_1 = Pirilampo("Luciferino", "Verde")
pirilampo_2 = Pirilampo("Vivido", "Amarelo")
```

Métodos de instância

- Além do método construtor, uma classe pode ter vários outros métodos, o que a fará permitir ao objeto poder realizar distintos comportamentos.

```
class Ovelha:
    def __init__(self, nome, criador):
        self.nome = nome
        self.criador = criador
        self.tamanho = 0
        self.idade = 0

    def falar (self, expressao, falar_ao_dono):
        if falar_ao_dono:
            print (f'{self.nome} diz "{expressao}" ao seu criador {self.criador}')
        else:
            print (f'{self.nome} diz "{expressao}"')


ovelha_dolly = Ovelha("Dolly", "Wilmut")
ovelha_dolly.falar("Meh meeeeh", True)
```

Método que permitirá à ovelha de falar

O nome do método deve começar ou ser um verbo (no infinitivo)!

É obrigatório ter o parâmetro "self" no início de cada método de instância!

Objeto do tipo ovelha a "falar ao seu dono"


```
C:\Users\userHP\AppData\Local\Programs\Python\Python39\p
Dolly diz "Meh meeeeh" ao seu criador Wilmut
```

Métodos de instância

Exercício 2

Aproveitando a classe do exercício anterior do pirilampo, acrescente os seguintes métodos:

- **Alterar a cor**: amarelo, verde e vermelho (apenas)
  - Apresentar: Antes tinha a cor X, agora tenho a cor Y
- **Alterar a intensidade de luz**: 0 a 100 (apenas)
  - Apresentar: Estou com a cor Y, na intensidade N %

Importante: sempre que se tente mudar de cor ou de intensidade para valores não permitidos, avise na consola dessa impossibilidade.

Faça com que os seus pirilampos mudem de cor e de intensidade!

Nota: Uma vez que a **cor** e a **intensidade** são características do pirilampo, guarde sempre nele próprio essa informação - deste modo, sempre que **alterar a cor** e seguidamente **alterar a intensidade da luz**, este poderá indicar a última cor que lhe foi atribuída.

- Cábula:

```
class Ovelha:
    def __init__(self, nome, criador):
        self.nome = nome
        self.criador = criador
        self.tamanho = 0
        self.idade = 0

    def falar (self, expressao, falar_ao_dono):
        if falar_ao_dono:
            print (f'{self.nome} diz "{expressao}" ao
seu criador {self.criador}')
        else:
            print (f'{self.nome} diz "{expressao}“’)


ovelha_dolly = Ovelha("Dolly", "Wilmut")
ovelha_dolly.falar("Meh meeeeh", True)
```

Métodos de instância

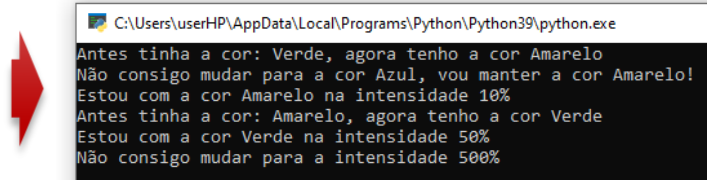
Exercício 2 (solução)

Classe

```
class Pirilampo:
    def __init__(self, nome, cor):
        self.nome = nome
        self.idade = 0
        self.tamanho = 0
        self.cor = cor
        self.intensidade_brilho = 0

    def alterar_cor(self, cor):
        cor = cor.capitalize() #compor o texto!
        if cor in ('Amarelo', 'Verde', 'Vermelho'):
            print (f"Antes tinha a cor: {self.cor}, agora tenho a cor {cor}")
            self.cor = cor
        else:
            print(f"Não consigo mudar para a cor {cor}, vou manter a cor {self.cor}!")

    def alterar_brilho(self, intensidade):
        if intensidade in range(101):
            print(f"Estou com a cor {self.cor} na intensidade {intensidade}%")
            self.intensidade_brilho = intensidade
        else:
            print(f"Não consigo mudar para a intensidade {intensidade}%")
```

Objetos a realizar os comportamentos para o qual foram programados:

```
pirilampo_1 = Pirilampo("Luciferino", "Verde")
pirilampo_2 = Pirilampo("Vivido", "Amarelo")

pirilampo_1.alterar_cor(“Amarelo")
pirilampo_1.alterar_cor("Azul")
pirilampo_1.alterar_intensidade(10)

pirilampo_2.alterar_cor("Verde")
pirilampo_2.alterar_intensidade(50)
pirilampo_2.alterar_intensidade(500)
```

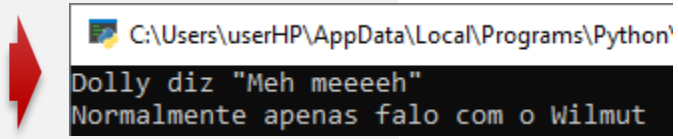


Métodos de instância

- Qualquer método poderá chamar um qualquer outro.

```
class Ovelha:
    def __init__(self, nome, criador):
        self.nome = nome
        self.criador = criador
        self.tamanho = 0
        self.idade = 0

    def falar(self, expressao, falar_ao_dono):
        if falar_ao_dono:
            print (f'{self.nome} diz "{expressao}" ao seu criador {self.criador}')
        else:
            print (F'{self.nome} diz "{expressao}"')
            self.dizer_algo_mais()

    def dizer_algo_mais(self):
        print (F"Normalmente apenas falo com o {self.criador}")


ovelhaDolly = Ovelha("Dolly", "Wilmut")
ovelhaDolly.falar("Meh meeeeh", False)
```

Importante: O “self” necessita de estar sempre presente a quando de chamarmos um outro método dentro da classe!



Métodos de instância

Exercício 3

Crie uma classe que possa representar uma tesoura.

Sabe-se que uma tesoura poderá nascer inicialmente com:

- **Usos disponíveis:** 100
- **Cor:** configurável

A tesoura deverá cortar o material que lhe for indicado:

- **Papel:** perde 1 dos seus usos
- **Plástico:** perde 20 dos seus usos
- **Metal:** perde 70 dos seus usos
- **Outro:** indicar como desconhecido (não é realizado corte)

  Sempre que a tesoura não tiver mais capacidade de corte, este não deve acontecer, sendo o utilizador informado para o efeito.

A tesoura deverá ainda possibilitar mostrar quantos usos ainda tem disponíveis. Esta funcionalidade deve ser sempre evocada depois de cada corte (caso tenha capacidade para o realizar).

▶ Cábula:

```
class Ovelha:
    def __init__(self, nome, criador):
        self.nome = nome
        self.criador = criador
        self.tamanho = 0
        self.idade = 0

    def falar (self, expressao, falar_ao_dono):
        if falar_ao_dono:
            print (F'{self.nome} diz "{expressao}" ao
seu criador {self.criador}')
        else:
            print (F'{self.nome} diz "{expressao}"')
            self.diz_algo_mais()

    def dizer_algo_mais(self):
        print (F"Normalmente apenas falo com o
{self.criador}")


ovelhaDolly = Ovelha("Dolly", "Wilmut")
ovelhaDolly.fala("Meh meeeeh", False)
```

Métodos de instância

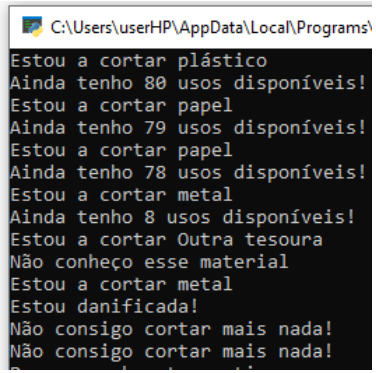
Exercício 3 (solução)

Classe

```
class Tesoura:
    def __init__(self, cor):
        self.cor = cor
        self.usos = 100

    def cortar(self, material):
        if self.usos <= 0:
            print ("Não consigo cortar mais nada!")
            return #sair do método neste instante!

        print (F"Estou a cortar {material}")
        material = material.lower()
        if material == "papel": self.usos -= 1
        elif material == "plástico": self.usos -= 20
        elif material == "metal": self.usos -= 70
        else:
            print("Não conheço esse material")
            return

        self.mostrar_usos()

    def mostrar_usos(self):
        if self.usos > 0: print(f"Ainda tenho {self.usos} usos disponíveis!")
        else: print("Estou danificada!")
```

Objetos a realizar os comportamentos para o qual foram programados:

```
tesoura = Tesoura("Vermelha")
tesoura.cortar("plástico")
tesoura.cortar("papel")
tesoura.cortar("papel")
tesoura.cortar("metal")
tesoura.cortar("Outra tesoura")
tesoura.cortar("metal")
tesoura.cortar("metal")
tesoura.cortar("metal")
```



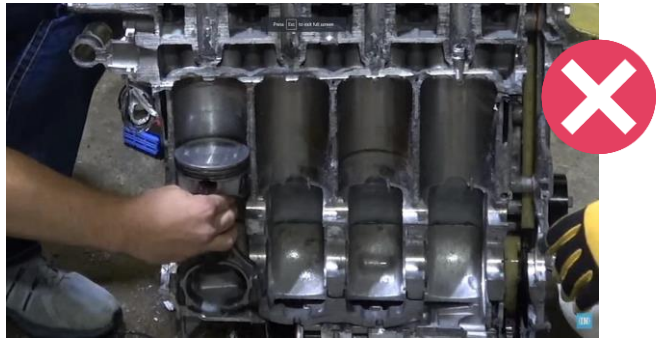
POO

Encapsulamento

Encapsulamento

O encapsulamento trata-se da forma como tornamos privado o acesso aos componentes do nosso objeto (métodos ou propriedades).

Assim como num carro, existem elementos ao qual termos acesso e outros ao qual não devemos ter - estes encontram-se “encapsulados”, não permitindo (por questões de segurança ou pelo correto funcionamento) o acesso direto ao utilizador.



Encapsulamento

Num carro, uma vez que não temos acesso direto ao motor, existem mecanismos externos que nos permitem que lho seja feito, de uma forma segura e sem colocar em causa o seu devido funcionamento.

! Forma insegura de controlar o motor

! Formas seguras de controlar o motor

Encapsulamento

Se as propriedades de uma classe estiverem públicas, a qualquer momento é possível alterar os seus valores de forma direta e não salvaguardada - isto que poderá resultar em comportamentos indesejados por parte dos objetos!

```
class Ovelha:
    def __init__(self, nome, criador):
        self.nome = nome
        self.criador = criador
        self.tamanho = 0
        self.idade = 0

    def falar(self, expressao):
        print(F'{self.nome} diz "{expressao}"')

ovelha_dolly = Ovelha("Dolly", "Wilmut")
ovelha_dolly.nome = "Chris"
print(f"Nome da ovelha: {ovelha_dolly.nome}")
ovelha_dolly.falar("Alteraram-me o nome diretamente sem eu permitir!")
```

Estas propriedades foram criadas como públicas

Uma propriedade pública será sempre acessível fora da classe, a partir do objeto! Isto poderá tornar instáveis os processos internos dos objetos, criando nestes comportamentos indesejáveis, o que resultará diretamente na instabilidade no programa!

```
Nome da ovelha: Chris
Chris diz "Alteraram-me o nome diretamente sem eu permitir!"
```

! Foi atribuído um valor a um atributo da classe de forma insegura!

Encapsulamento

O modo correto é privar sempre todas as propriedades!

Para privar uma propriedade, no código da classe devemos colocar-lhe o prefixo "__" (2 underscores seguidos).

```
class Ovelha:
    def __init__(self, nome, criador):
        self.__nome = nome
        self.__criador = criador
        self.__tamanho = 0
        self.__idade = 0

    def falar(self, expressao):
        print(f'{self.__nome} diz "{expressao}"')
```

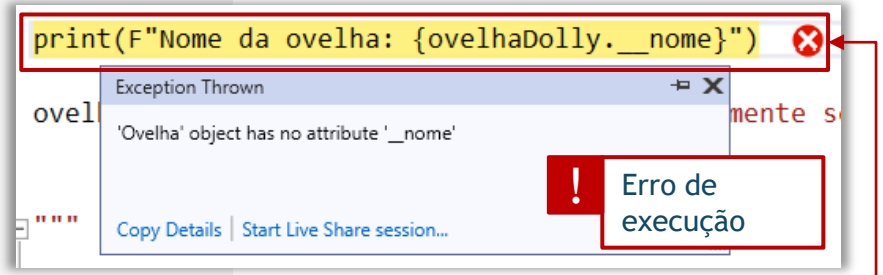
Propriedades definidas como privadas

Sempre que precisarmos de aceder a uma propriedade privada (dentro da própria classe), necessitamos de respeitar o prefixo!

Encapsulamento

Ao aceder publicamente a uma propriedade privada a partir de um objeto, o interpretador irá reagir com um erro a informar a impossibilidade do acesso!

```
class Ovelha:
    def __init__( self, nome, criador ):
        self.__nome = nome
        self.__criador = criador
        self.__tamanho = 0
        self.__idade = 0

    def falar(self, expressao):
            print (f'{self.__nome} diz "{expressao}"')

ovelha_dolly = Ovelha("Dolly", "Wilmut")
print(f"Nome da ovelha: {ovelha_dolly.__nome}")
```

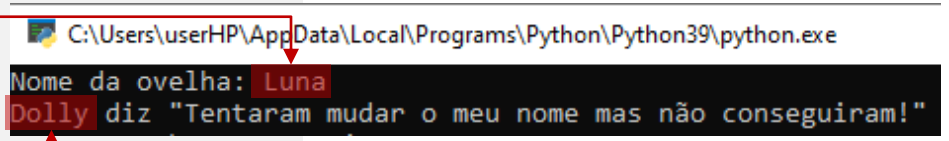


Encapsulamento

Porém se se tentar alterar o valor de uma propriedade privada de um objeto, esta ação não resultará diretamente em um erro de execução. A propriedade ficará indiretamente atribuída à "superfície do objeto".

É importante salientar que esta atribuição indireta de propriedades, embora não afete diretamente as propriedades privadas do objeto, poderá originar problemas futuros e, por isso, nunca deve ser realizada.

```
class Ovelha:
    def __init__(self, nome, criador):
        self.__nome = nome
        self.__criador = criador
        self.__tamanho = 0
        self.__idade = 0

    def falar(self, expressao):
        print (f'{self.__nome} diz "{expressao}"')

ovelha_dolly = Ovelha("Dolly", "Wilmut")

ovelha_dolly.__nome = "Luna"
print(f"Nome da ovelha: {ovelha_dolly.__nome}")

ovelha_dolly.falar("Tentaram mudar o meu nome mas não conseguiram!")
```



Apesar da tentativa e não tendo resultado em um erro de execução, o objeto manteve o valor original da sua propriedade.

Esta variável acabou por ser criada mas não no contexto interno (que se mantem privado) do objeto.

Encapsulamento

O mesmo se aplica para privar o acesso a métodos, onde também se coloca o prefixo “__” (2 underscores seguidos).

A tentativa pública de acesso a um método privado resultará num erro de execução.

```
class Ovelha:
    def __init__(self, nome, criador):
        self.__nome = nome
        self.__criador = criador
        self.__tamanho = 0
        self.__idade = 0

    def __falar(self, expressao):
        print(f'{self.__nome} diz "{expressao}"')

ovelha_dolly = Ovelha("Dolly", "Wilmut")
ovelha_dolly.__falar("Meeeeeh")
```

Encapsulamento

Exercício 4

Crie uma classe que possa representar uma caneta.

Sabe-se que uma caneta poderá nascer inicialmente com:

- **Quantidade de tinta:** 100 metros
- **Quantidade de utilizações:** zero
- **Marca:** configurável
- **Cor:** configurável

Crie ainda duas canetas provenientes da sua classe:

- “Uni-ball” com a cor preto e “BIC” com a cor azul

Não se esqueça de privar o acesso ás propriedades!

Por fim: tente mostrar a cor das canetas acendendo diretamente às propriedades das mesmas – avalie o resultado.

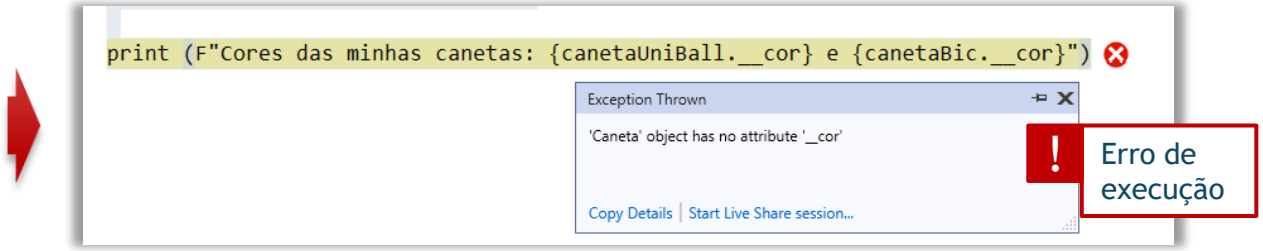
- Cábula:

```
class Ovelha:
    def __init__(self, nome, criador):
        self.__nome = nome
        self.__criador = criador
        self.__tamanho = 0
        self.__idade = 0

    def __falar(self, expressao):
            print(f'{self.__nome} diz "{expressao}"')

ovelha_dolly = Ovelha("Dolly", "Wilmut")
```

Encapsulamento

Exercício 4

```
class Caneta:
    def __init__(self, marca, cor):
        self.__qt_tinta = 100
        self.__qt_usos = 0
        self.__marca = marca
        self.__cor = cor

caneta_uniball = Caneta("Uni-ball", "Preto")
caneta_bic = Caneta("BIC", "Azul")

print (F"Cores das minhas canetas: {caneta_uniball.__cor} e {caneta_bic.__cor}")
```



POO

Getters e Setters

Encapsulamento

Privado o acesso às propriedades, fica impossibilitado publicamente o seu acesso.

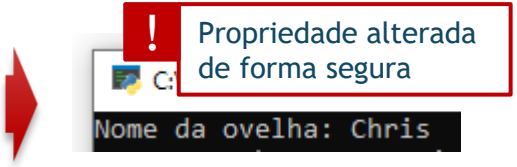
O acesso às propriedades que se reservam privadas, requer a criação de métodos públicos que asseguram um controlo mais adequado à leitura e modificação dos seus dados.

Estes métodos são chamados de Setters e Getters, e permitem atribuir \ devolver (respetivamente) valores de uma propriedade.

```
class Ovelha:
    def __init__(self, nome, criador):
        self.__nome = nome
        self.__criador = criador
        self.__tamanho = 0
        self.__idade = 0

    def get_nome(self):
        return self.__nome

    def set_nome(self, value):
        self.__nome = value

ovelhaDolly = Ovelha("Dolly", "Wilmut")
ovelhaDolly.set_nome("Chris") #alterar o nome de forma segura
print(F"Nome da ovelha: {ovelhaDolly.get_nome()}") #buscar o nome de forma segura
```

Getter: Retorna o valor da propriedade

Setter: Recebe um valor por parâmetro e atribui-o à propriedade



Encapsulamento

Exercício 5

Aproveitando a classe anterior da Caneta, crie a possibilidade de externamente:

- A cor poder ser alterada e acedida.
- Podermos saber qual a marca da caneta.

Seguidamente a partir dos objetos altere a cor das suas canetas e mostre as suas marcas e cores atuais.

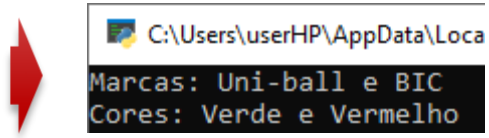
- Cábula:

```
class Ovelha:
    def __init__( self, nome, criador ):
        self.__nome = nome
        self.__criador = criador
        self.__tamanho = 0
        self.__idade = 0

    def get_nome(self):
        return self.__nome

    def set_nome(self, value):
        self.__nome = value

ovelhaDolly = Ovelha("Dolly", "Wilmut")
ovelhaDolly.set_nome("Chris")
print(F"Nome da ovelha: {ovelhaDolly.get_nome()}")
```

Encapsulamento

Exercício 5 (solução)

```
class Caneta:
    def __init__(self, marca, cor):
        self.__qt_tinta = 100
        self.__qt_usos = 0
        self.__marca = marca
        self.__cor = cor

    def set_cor(self, cor):
        self.__cor = cor

    def get_cor(self):
        return self.__cor

    def get_marca(self):
        return self.__marca

caneta_uniball = Caneta("Uni-ball", "Preto")
caneta_bic = Caneta("BIC", "Azul")

caneta_uniball.set_cor("Verde")
caneta_bic.set_cor("Vermelho")
print(f"Marcas: {caneta_uniball.get_marca()} e {caneta_bic.get_marca()}")
print(f"Cores: {caneta_uniball.get_cor()} e {caneta_bic.get_cor()}")
```



Encapsulamento

Em algumas linguagens de programação, a utilização de métodos é a única forma de controlar o acesso seguro às propriedades.

No entanto, o Python oferece um método alternativo, permitindo ao objeto tratar um Getter e Setter como se este se tratasse de uma propriedade, apesar de continuar a ser um método e oferecer todas as vantagens de um. Para isto são utilizados os declaradores "**@property**" e "**@*****.setter**".

Classe

```
class Ovelha:
    def __init__(self, nome, criador):
        self.__nome = nome
        self.__criador = criador
        self.__tamanho = 0
        self.__idade = 0

    @property
    def nome(self):
        return self.__nome

    @nome.setter
    def nome(self, valor):
        self.__nome = valor
```

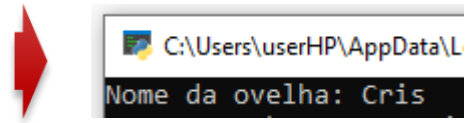
Getter

Setter

Objeto a utilizar os Getters e Setters

```
ovelha_dolly = Ovelha("Dolly", "Wilmut")

ovelha_dolly.nome = "Cris"
print("Nome da ovelha:", ovelha_dolly.nome)
```

Este tipo de Getters e Setters tratam-se como se fossem variáveis convencionais (não têm parenteses e podem ser atribuídos valores às mesmas).



! Getters e Setters não deixam de ser métodos, mas para o objeto, comportam-se como se se tratassem de propriedades com aceso público!

Encapsulamento

Regras para a criação de propriedades com Getter e Setter:

```
class Ovelha:
    def __init__(self, nome, criador):
        self.__nome = nome
        self.__criador = criador
        self.__tamanho = 0
        self.__idade = 0

    @property
    def nome(self):
        return self.__nome

    @nome.setter
    def nome(self, valor):
        self.__nome = valor

ovelha_dolly = Ovelha("Dolly", "Wilmut")

ovelha_dolly.nome = "Cris"
print("Nome da ovelha:", ovelha_dolly.nome)
```

! Declaradores obrigatórios! (Os declaradores são definidos com um @).

Nota: Não pode existir um Setter sem o respetivo Getter, mas pode existir um Getter sem o respetivo Setter (no método convencional esta limitação não existe)

Encapsulamento

Regras para a criação de propriedades com Getter e Setter:

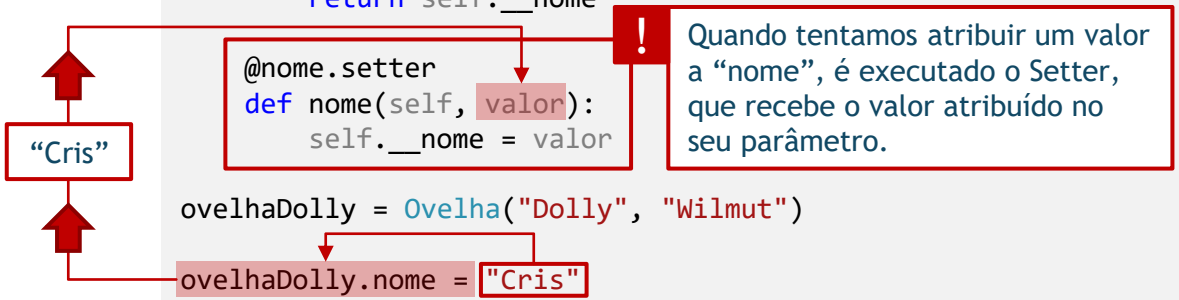
```
class Ovelha:
    def __init__( self, nome, criador ):
        self.__nome = nome
        self.__criador = criador
        self.__tamanho = 0
        self.__idade = 0

    @property
    def nome(self):
        return self.__nome

    @nome.setter
    def nome(self, valor):
        self.__nome = valor

ovelhaDolly = Ovelha("Dolly", "Wilmut")

ovelhaDolly.nome = "Cris"
print("Nome da ovelha:", ovelhaDolly.nome)
```

! Necessitam de ter o mesmo nome! Inclui-se também a definição no declarador do Setter.
Nota: pode escolher qualquer nome que desejar!

Encapsulamento

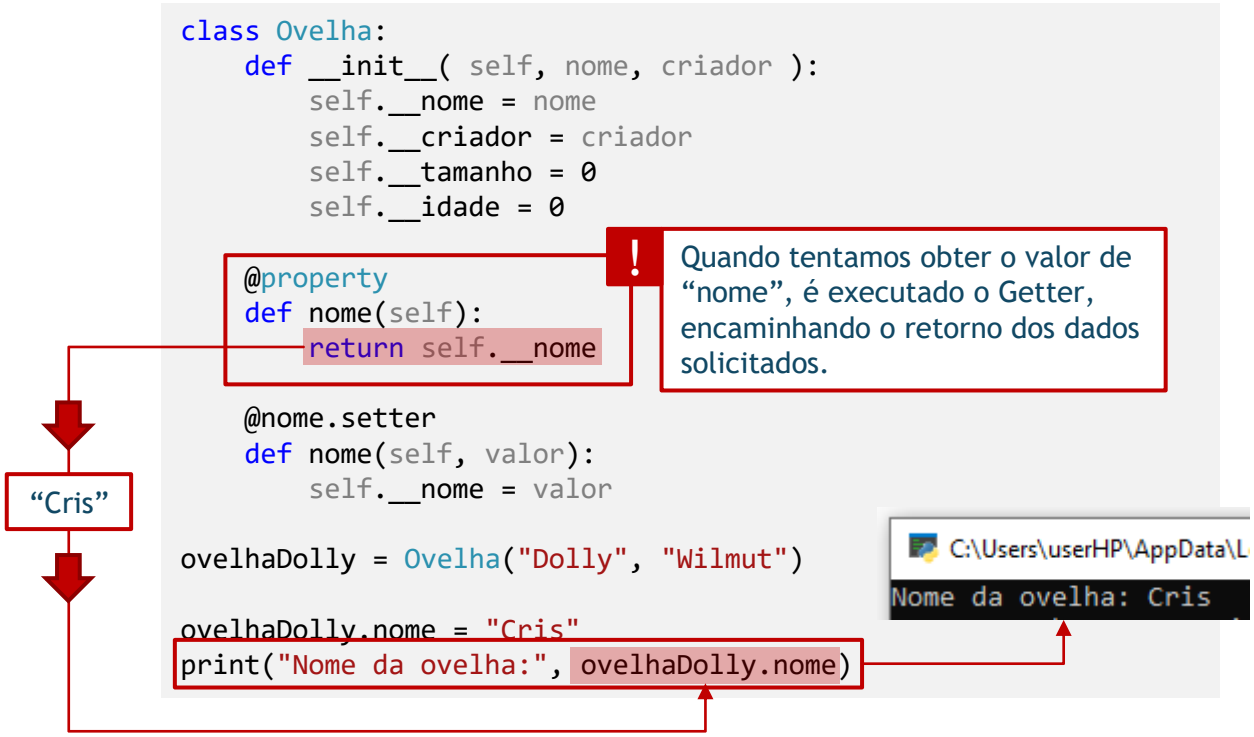
Regras para a criação de propriedades com Getter e Setter:

```
class Ovelha:
    def __init__( self, nome, criador ):
        self.__nome = nome
        self.__criador = criador
        self.__tamanho = 0
        self.__idade = 0

    @property
    def nome(self):
        return self.__nome

    @nome.setter
    def nome(self, valor):
        self.__nome = valor

ovelhaDolly = Ovelha("Dolly", "Wilmut")

ovelhaDolly.nome = "Cris"
print("Nome da ovelha:", ovelhaDolly.nome)
```



Encapsulamento

Regras para a criação de propriedades com Getter e Setter:



Encapsulamento

Exercício 6

Crie uma classe que possa representar uma mola.

Sabe-se que uma mola poderá nascer inicialmente com:

- **Extensão máxima:** configurável
- **Extensão atual:** deverá ficar a 50% da máxima

Crie os Getter e Setter para a extensão atual (utilizando declaradores).

Nota: A atribuição da extensão atual nunca poderá ficar abaixo de zero ou exceder o valor máximo – caso aconteça, corrija o problema atribuindo o valor mais alto ou baixo correspondendo ao limite excedido.

- Cábula:

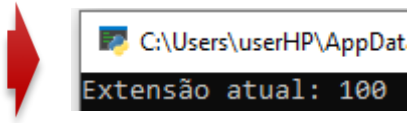
```
class Ovelha:
    def __init__( self, nome, criador ):
        self.__nome = nome
        self.__criador = criador
        self.__tamanho = 0
        self.__idade = 0

    @property
    def nome(self):
        return self.__nome

    @nome.setter
    def nome(self, valor):
        self.__nome = valor

ovelha_dolly = Ovelha("Dolly", "Wilmut")

ovelha_dolly.nome = "Cris"
print("Nome da ovelha:", ovelha_dolly.nome)
```

Encapsulamento

Exercício 6

```
class Mola:
    def __init__(self, maximo):
        self.__ext_maxima = maximo
        self.__ext_atual = maximo / 2

    @property
    def ext_atual(self):
        return self.__ext_atual

    @extAtual.setter
    def extAtual(self, valor):
        if valor < 0:
            self.__ext_atual = 0
        elif valor > self.__ext_maxima:
            self.__ext_atual = self.__ext_maxima
        else:
            self.__ext_atual = valor

mola = Mola(100)
mola.ext_atual = 5000
print("Extensão atual:", mola.ext_atual)
```

! Note que apesar de ter sido atribuída uma extensão de 5000, o valor máximo nunca foi excedido porque o Setter teve esse cuidado!

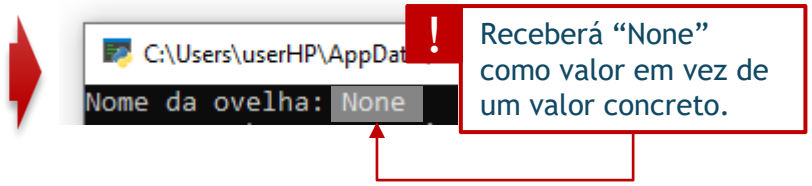


Encapsulamento

Para criar um Setter sem o respetivo Getter, pode ser realizada a seguinte estratégia:

```
class Ovelha:
    def __init__(self, nome, criador):
        self.__nome = nome
        self.__criador = criador
        self.__tamanho = 0
        self.__idade = 0

    @property
    def nome(self):
        return None

    @nome.setter
    def nome(self, valor):
        self.__nome = valor

ovelha_dolly = Ovelha("Dolly", "Wilmut")

ovelha_dolly.nome = "Cris"
print("Nome da ovelha:", ovelha_dolly.nome)
```

! Retornar "None"!
"None" equivalente em outras linguagens a "Null". Este representa a inexistência de um valor.



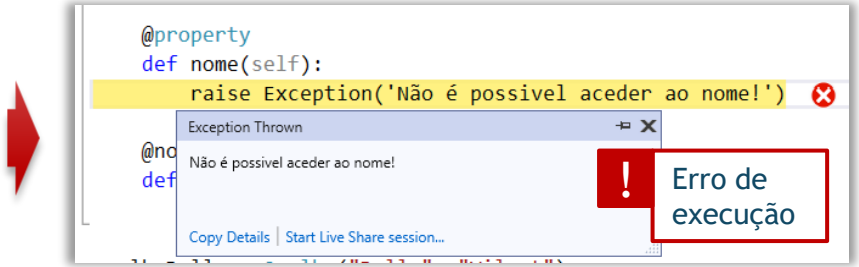
! Receberá "None" como valor em vez de um valor concreto.

Encapsulamento

… ou impossibilite a continuidade da execução do programa despoletando um erro (através do levantando uma exceção)!

```
class Ovelha:
    def __init__( self, nome, criador ):
        self.__nome = nome
        self.__criador = criador
        self.__tamanho = 0
        self.__idade = 0

    @property
    def nome(self):
        raise Exception('Não é possivel aceder ao nome!')

    @nome.setter
    def nome(self, valor):
        self.__nome = valor

ovelha_dolly = Ovelha("Dolly", "Wilmut")

ovelha_dolly.nome = "Cris"
print("Nome da ovelha:", ovelha_dolly.nome)
```

! Mate a execução do programa com um erro!



POO

Exceções

Exceções

Existem dois tipos de erros: os que são detetados antes e o que são detetados durante da execução de um programa:

Runtime error: Acontece durante a execução de um programa, fazendo com que este deixe de funcionar (grande possibilidade de estarem associados a erros lógicos) – o IDE não deteta estes erros. Estes erros acontecem já durante a execução do programa devido a algo que não se esperava acontecer!

```
while True:
    print ("Olá mundo")
```

! Ciclo infinito

```
valor = int(input("Por quanto quer dividir?"))
print (2 / valor)
```

! Divisão por zero poderá ocorrer

Sintax error: Erros cometidos pelo programador que desrespeitam a gramática da linguagem de programação, fazendo com que o programa não seja compilado - o interpretador deteta estes erros

```
minhaString = "Olá mundo"

if minhaString == "texto"
print minhaString
else:
```

! Problemas:
- Faltam “:” no fim do if e do else
- Falta Indentação no if
- Falta código no else

Exceções

Existem ainda os erros lógicos e de tipo de dados:

Logic error: Erro lógico (O programa funciona mas produz resultados inesperados) – detetado apenas durante a execução do programa e somente por humanos

```
preco = float(input("Quanto custa o produto:\n"))
print (F"Produto com iva: {preco:.2f}")
```

! Faltou calcular o IVA

Type error: Tipo de dados incompatíveis ou inesperados - detetado apenas durante a execução do programa

```
meuInteiro = 3
minhaString = "4"

print(meuInteiro + minhaString)
```

! Não é possível concatenar ou somar Inteiros a Strings (não são tipos de dados compatíveis)

Exceções

E ainda os erros de nomes:

Name Error: Acontece quando o programa acede a uma variável ao qual não lhe foi previamente atribuído um valor (ou seja a variável não se encontra inicializada naquele momento) - detetado apenas durante a execução do programa

```
while continua != "s":
    continua = input("Pretende continuar S\\N?")
```

! Falta de inicialização da variável para que possa ter um valor válido para conseguir ser usada no while.

TutorialsTeacher

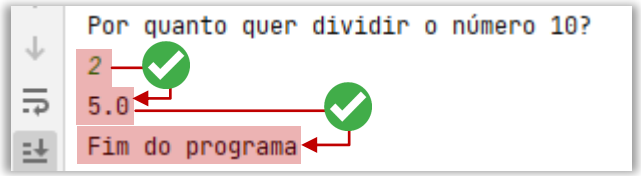
Lista detalhada de tipos de erros do Python:
https://www.tutorialsteacher.com/python/error-types-in-python

Tratamento de exceções

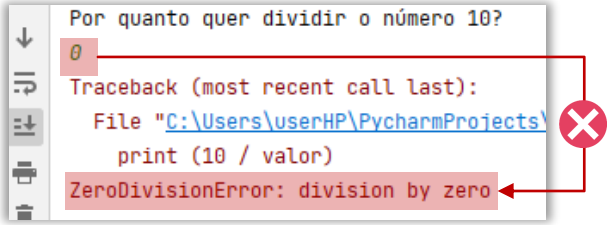
Existem portanto erros que apenas podem ocorrer durante a execução do programa e podem por exemplo depender do input do utilizador.

Dividir por zero é um desses casos:

```
valor = int(input("Por quanto quer dividir o número 10?\n"))
print(10 / valor)

print("Fim do programa")
```



! O input do utilizador não invalidou a execução do código, sendo possível chegar até ao fim da execução do mesmo.

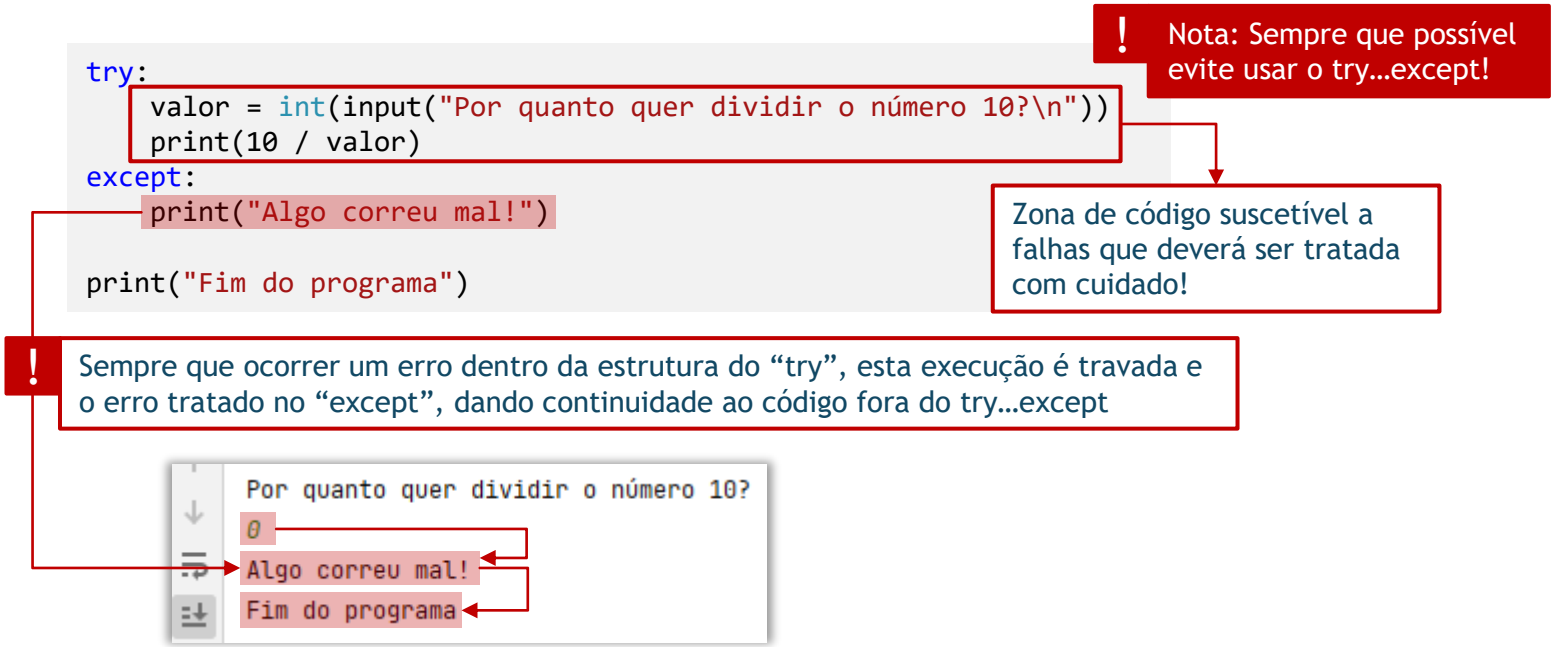


! O input do utilizador abortou a execução do código (foi realizada uma divisão por zero).

Tratamento de exceções

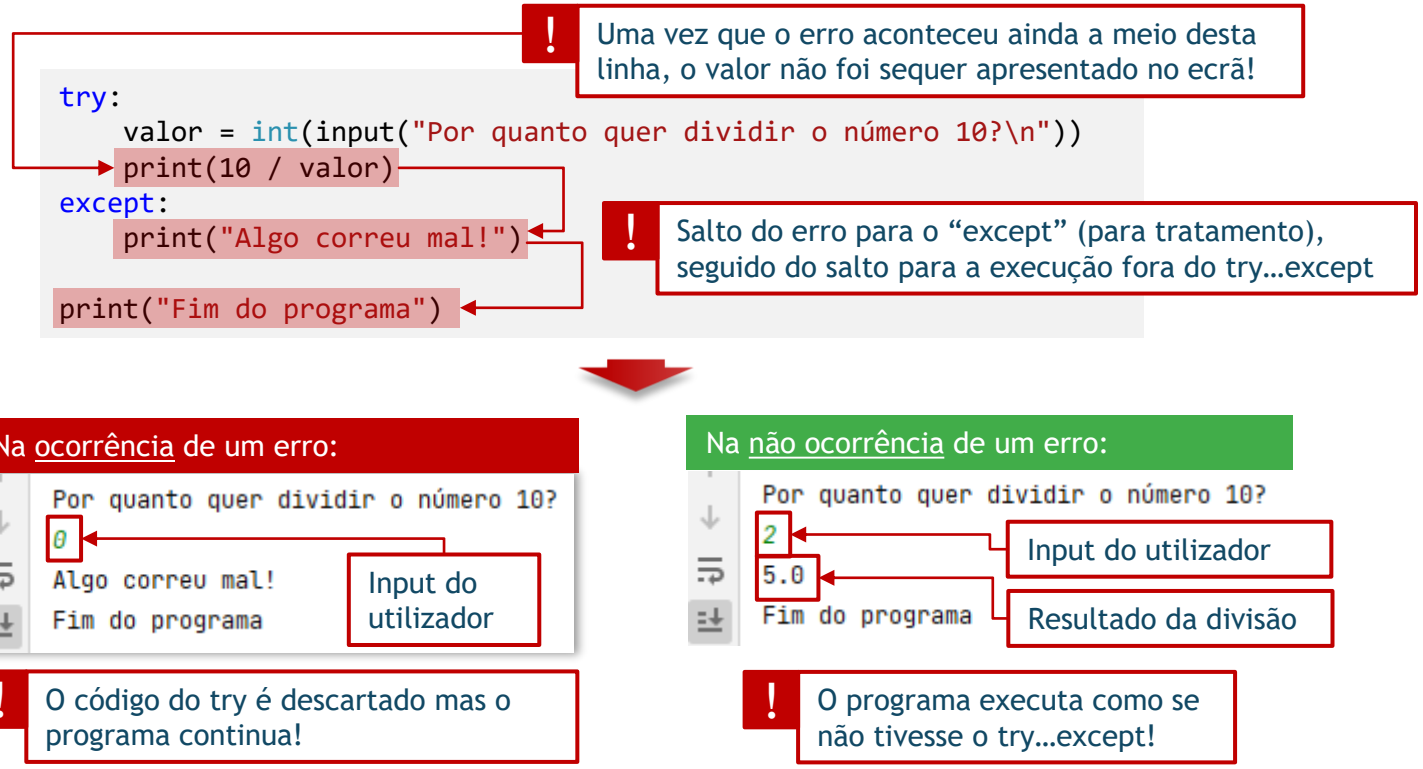
A grande maioria das linguagens de programação (assim como o Python) oferece tratamento de exceções.

Exceções tratam-se de erros que ocorrem durante a execução do programa – excluem-se portanto os erros de sintaxe!



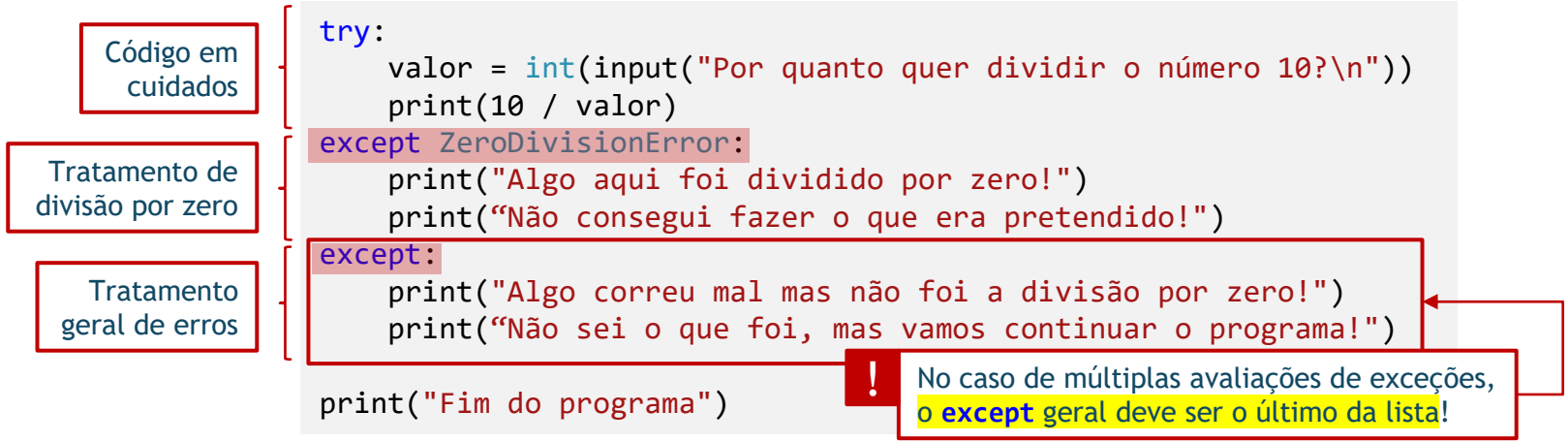
Tratamento de exceções

Note que quando um erro acontece dentro do try, o código a seguir ao erro (ainda dentro do try) não será executado:



Tratamento de exceções

É ainda possível enumerar vários blocos de exceções para o mesmo try, para que se possam tratar de formas distintas:



Tratamento de exceções

Ainda dentro do try, é possível avaliar situações onde não ocorram erros acrescentando no fim um “else”:

Código em cuidados

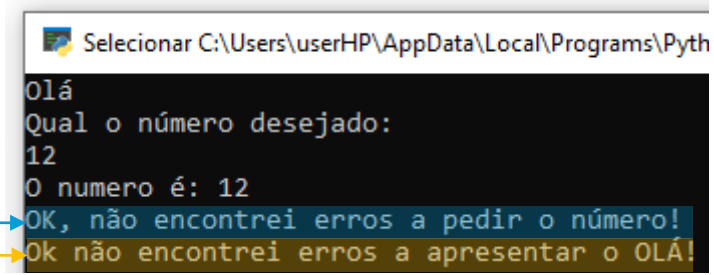
```
try:
  print("Olá")
except:
  print("Aconteceu algo de errado aqui!")
else:
  print("Ok não encontrei erros!")
```

! Este “else” associa-se ao try para ser executado em caso não aconteça nenhum erro. Necessita de ser o último de todos!

Tratamento de exceções

É ainda possível encadear try…excepts em qualquer zona destes, uma vez que qualquer local dentro de um "try:", "except:" ou "else:" poderá receber qualquer outro conjunto de instruções, incluindo outro try…except:

```
try:
  print("Olá")
  try:
      num = int(input("Qual o número desejado:\n"))
      print("O numero é:", num)
  except:
      print("Aconteceu algo de errado a pedir o numero!")
  else:
      print("OK, não encontrei erros a pedir o número!")
except:
  print("Aconteceu algo de errado ao apresentar o OLÁ!")
else:
  print("Ok não encontrei erros a apresentar o OLÁ!")
```



Tratamento de exceções

Uma vez que é possível simular exceções, os objetos também poderão emitir as mesmas, impedindo a continuação do programa:

```
class Ovelha:
    def __init__( self, nome, criador ):
        self.__nome = nome
        self.__criador = criador
        self.__tamanho = 0
        self.__idade = 0

    def set_tamanho(self, tamanho):
        if tamanho <= 0:
            raise Exception("Tamanho inválido!")

        self.__tamanho = tamanho

ovelhaDolly = Ovelha("Dolly", "Wilmut")
ovelhaDolly.set_tamanho(0)
```

1

2

```
def setTamanho(self, tamanho):
    if tamanho <= 0:
        raise Exception("Tamanho inválido!")
```

3

Exception Thrown

Tamanho inválido!

Resultado de uma simulação de um erro (o programa aborta)

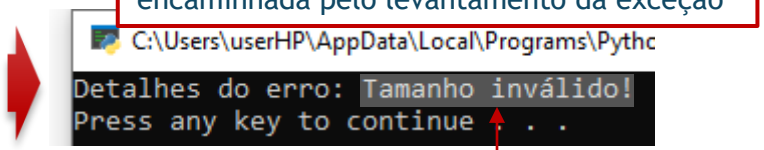
Tratamento de exceções

Nestes casos, podemos também usar o try…except para a resolução do mesmo:

```
class Ovelha:
    def __init__( self, nome, criador ):
        self.__nome = nome
        self.__criador = criador
        self.__tamanho = 0
        self.__idade = 0

    def set_tamanho(self, tamanho):
        if tamanho <= 0:
            raise Exception("Tamanho inválido!")

        self.__tamanho = tamanho

ovelha_dolly = Ovelha("Dolly", "Wilmut")

try:
    ovelha_dolly.set_tamanho(0)

except Exception as ex:
    print("Detalhes do erro:", ex)
```

! Esta linha de código não será atingida a quando do levantamento da exceção!

! É possível obter a mensagem que foi encaminhada pelo levantamento da exceção



Tratamento de exceções

É ainda possível fazer o levantamento de exceções de tipos específicos:

```
class Ovelha:
    def __init__( self, nome, criador ):
        self.__nome = nome
        self.__criador = criador
        self.__tamanho = 0
        self.__idade = 0

    def set_tamanho(self, tamanho):
        if tamanho <= 0:
            raise ValueError("Tamanho inválido!")
        self.__tamanho = tamanho

ovelha_dolly = Ovelha("Dolly", "Wilmut")
try:
    ovelha_dolly.set_tamanho(-20)
except ValueError as ex:
    print ("Detalhes do erro:", ex)
```

TutorialsTeacher

Lista detalhada de tipos de erros do Python:
https://www.tutorialsteacher.com/python/error-types-in-python

! Indicado como "ValueError"

! Tratado como como "ValueError"

Tratamento de exceções

Exercício 7

Crie uma classe que possa representar uma bola.

Permita que a sua bola possa ser construída com:

- Pressão máxima
- Pressão mínima
- Pressão atual

Não permita que a pressão máxima seja inferior à mínima (e vice-versa). Não permita ainda que a pressão atual saia dos limites máximos e mínimos - levante exceções para ambos os casos.

Faça a bola saltar, perdendo 10 pontos por cada salto (da pressão atual) – informe que a bola está a saltar.

Quando a bola estiver abaixo da pressão mínima, esta não deve saltar indicando um erro (levante exceções).

Trate as exceções do seu código diretamente com o objeto!

- Cábula:

```
class Ovelha:
    def __init__(self, nome, criador):
        self.__nome = nome
        self.__criador = criador
        self.__tamanho = 0
        self.__idade = 0

    def set_tamanho(self, tamanho):
        if tamanho <= 0:
            raise ValueError("Tamanho inválido!")
        self.__tamanho = tamanho

ovelha_dolly = Ovelha("Dolly", "Wilmut")
try:
    ovelha_dolly.set_tamanho(-20)
except ValueError as ex:
    print ("Detalhes do erro:", ex)
```

```
try:
  print("Olá")
except:
  print("Aconteceu algo de errado aqui!")
else:
  print("Ok não encontrei erros!")
```

Tratamento de exceções

Exercício 7 (solução)

Classe

```
class Bola:
    def __init__(self, pres_min, pres_max, pres_atual):
        if pres_max < pres_min:
            raise ValueError("Pressão máxima não pode ser inferior à mínima")

        if pres_atual < pres_min or pres_atual > pres_max:
            raise ValueError("Pressão atual está fora da mínima e máxima!")

        self.__pres_min = pres_min
        self.__pres_max = pres_max
        self.__pres_atual = pres_atual

    def salta(self):
        if self.__pres_atual < self.__pres_min:
            raise Exception ("A bola não pode saltar: Pouca pressão!")

        print("Bola a saltar!")
        self.__pres_atual -= 10
```

Bola em funcionamento:

```
try:
    bola = Bola(10, 50, 20)
except ValueError as ex:
    print("Erro: ", ex)
else:
    try:
        bola.salta()
        bola.salta()
        bola.salta()
        bola.salta()
    except Exception as ex:
        print("Erro: ", ex)
```

! Pela pressão da bola ter ficado baixa, uma exceção foi lançada nesta instrução, resultando no imediato descartar da continuidade da execução de código dentro do "try"!

```
C:\Users\userHP\AppData\Local\Programs\Python\Python39\python.exe
Bola a saltar!
Bola a saltar!
Erro:  A bola não pode saltar: pressão abaixo da mínima!
```

POO

Herança

Herança

A herança trata-se da capacidade de uma classe filha poder herdar (receber) herdar atributos (propriedades) e comportamentos (métodos) de uma classe mãe.

Mãe e filha:
Apesar da filha ter os seus atributos (propriedades) e comportamentos (métodos) próprios, ela herdou ainda dos atributos da mãe: a cor da pele, cabelo e olhos; o formato dos olhos, nariz e queixo. E inclusive parte da personalidade (comportamentos) da mãe.
Ela foi “construída” à imagem da mãe!

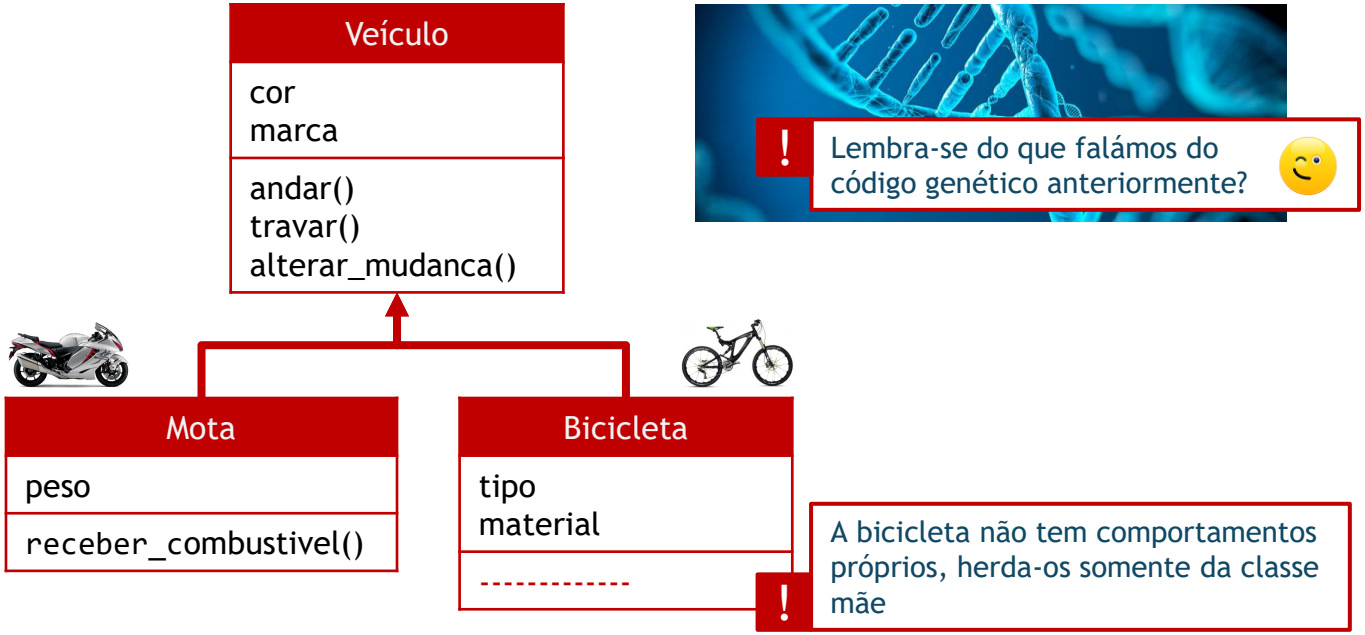
Herança

Imagine as classes moto e bicicleta com os seguintes atributos e comportamentos:

Atributos	Cor	Atributos comuns	Cor
	Marca		Marca
	Peso		Tipo
			Material
Comportamentos	Andar	Comportamentos comuns	Andar
	Travar		Travar
	Alterar mudança		Alterar mudança
	Receber combustível		

Herança

Por forma a simplificarmos o nosso código, devemos criar uma hierarquia entre classes, onde a classe veículo (classe mãe) terá todas as características e comportamentos comuns que as classes filhas necessitam, e que poderão herdar desta:



Herança

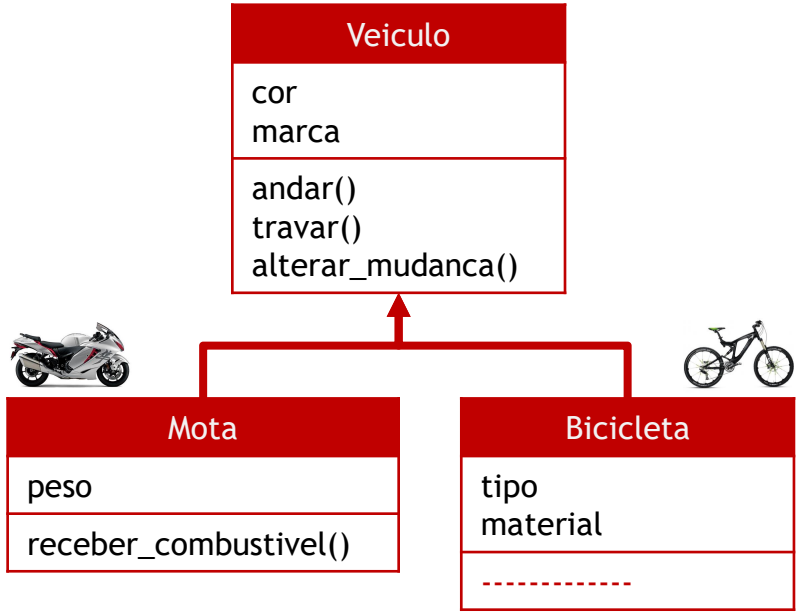
Exercício 8

Crie um projeto novo e nele crie as 3 classes representadas ao lado (em ficheiros separados, cada um deles com o mesmo nome da classe).

Importante:

- Todos os atributos são configuráveis (e privados) aquando da construção do objeto.
- Crie os Getters e Setters
- Nos métodos coloque apenas um print a indicar a tarefa que está a realizar, exemplo:

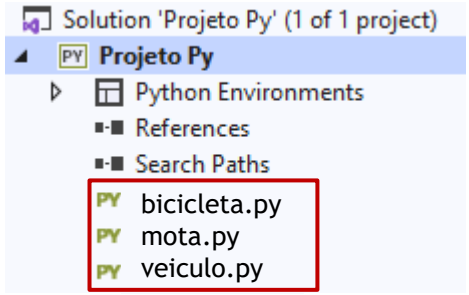
```
def andar(self):
   print ("Estou a andar")
```



Herança

Exercício 8 (solução)

Ficheiros do projeto:



Veículo

```
class Veiculo:
    def __init__(self, cor, marca):
        self.__cor = cor
        self.__marca = marca

    def set_cor(self, cor):
        self.__cor = cor

    def get_cor(self):
        return self.__cor

    def get_marca(self):
        return self.__marca

    def set_marca(self, marca):
        self.__marca = marca

    def andar(self):
        print ("Estou a andar")

    def travar(self):
        print("Estou a travar")

    def alterar_mudanca(self):
        print("Estou a alterar mudanças")
```

Mota

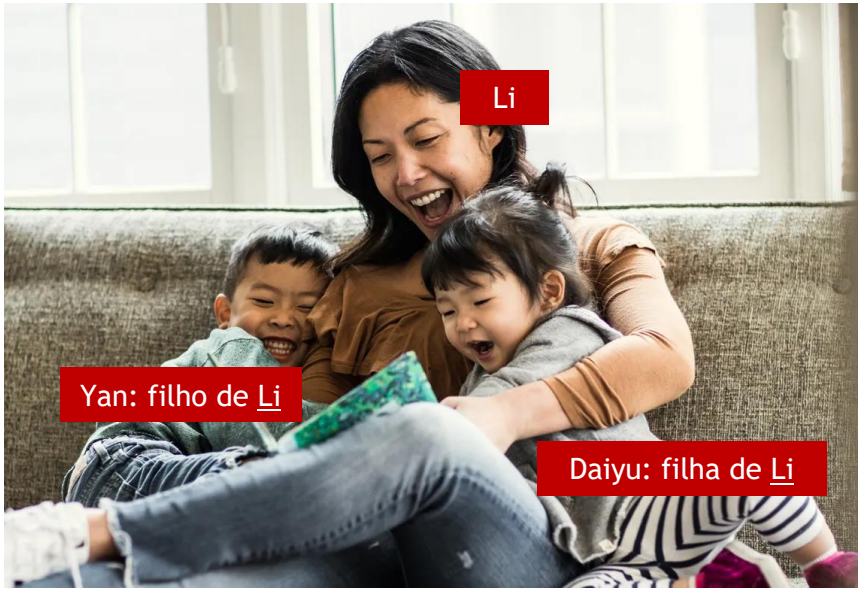
```
class Mota:
    def __init__(self, peso, cilindrada):
        self.__peso = peso
        self.__cilindrada = cilindrada

    def set_peso(self, peso):
        self.__peso = peso

    def get_peso(self):
        return self.__peso

    def receber_combustivel(self):
        print("Estou a receber combustível")
```

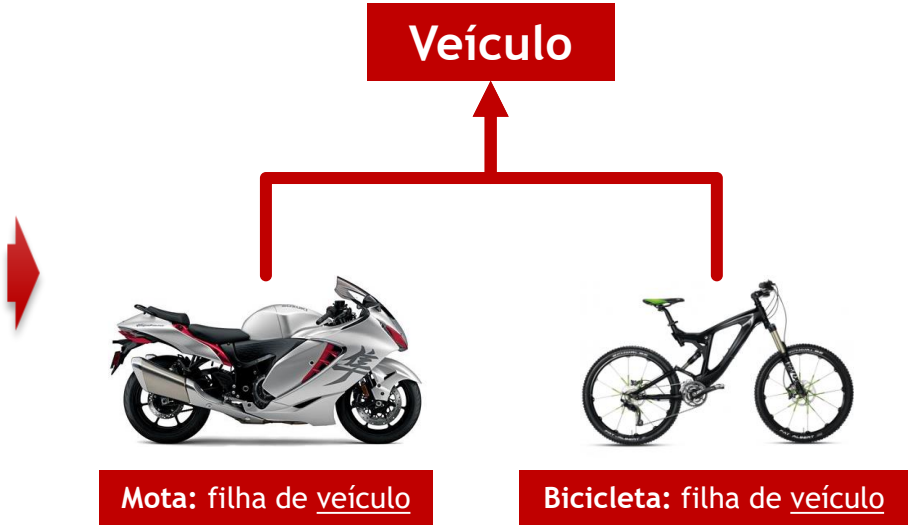
Bicicleta

```
class Bicicleta:
    def __init__(self, tipo, material):
        self.__tipo = tipo
        self.__material = material
```

Herança

Para que as classes possam ter hierarquia, necessitamos de lhes indicar quem é a sua mãe:





Herança

Para que as classes possam ter hierarquia, necessitamos de lhes indicar de quem elas são filhas:

Indicação da classe mãe

```
from veiculo import *

class Mota (Veiculo) :
    def __init__(self, peso, cilindrada):
        self.__peso = peso
        self.__cilindrada = cilindrada

    def set_peso(self, peso):
        self.__peso = peso

    def get_peso(self):
        return self.__peso

    def receber_combustivel(self):
        print("Estou a receber combustível")
```

Mota

! Importar do ficheiro Veículo.py a classe Veiculo (para que possa ser abaixo usada na hierarquia)

Sintaxe:
`from ficheiro import todo o conteúdo`

Herança

… e necessitamos de fazer algo a este construtor, para que ao construir um objeto a partir da classe filha, ele possa receber todos os atributos:

Mota

```
from veiculo import *

class Mota (Veiculo):
    def __init__(self, cor, marca, peso, cilindrada):
        super().__init__(cor, marca)
        self.__peso = peso
        self.__cilindrada = cilindrada

    def set_peso(self, peso):
        self.__peso = peso

    def get_peso(self):
        return self.__peso

    def receber_combustivel(self):
        print("Estou a receber combustível")
```

! Sempre que queiramos chamar algo da mãe, devemos usar a superclasse (classe mãe)!
Neste caso chamámos o construtor da classe mãe para que possamos atribuir o que está em falta do objeto mota enquanto veículo.
Nota: A evocação do método construtor da superclasse deve ser a primeira linha no método construtor da classe filha.

Herança

Posto isto, já podemos criar o objeto com as devidas propriedades e chamar os métodos da classe filha e mãe no mesmo objeto (uma vez que são partilhados).

Mota

```
from Veiculo import *

class Mota (Veiculo):
    def __init__(self, cor, marca, peso, cilindrada):
        super().__init__(cor, marca)
        self.__peso = peso
        self.__cilindrada = cilindrada

    def set_peso(self, peso):
        self.__peso = peso

    def get_peso(self):
        return self.__peso

    def receber_combustivel(self):
        print("Estou a receber combustível")

mota = Mota("Vermelho", "Honda", 100, 500)
mota.receber_combustivel()
print("Marca da mota:", mota.get_carca())
```

Construtor com propriedades da classe mota e veículo

Método da classe mota

Método da classe mãe (veículo)
Os métodos da superclasse, desde que públicos, podem ser chamados diretamente no objeto.

Herança

Exercício 9

Acrescente ao exercício anterior a possibilidade da bicicleta herdar da classe veículo.

Crie um novo ficheiro chamado “Main.py”, nele crie uma bicicleta e coloque-a a andar e a travar (comportamentos que esta deverá herdar da classe veículo).

Nota: no ficheiro da classe da bicicleta não deverá haver mais nada além da classe em si!

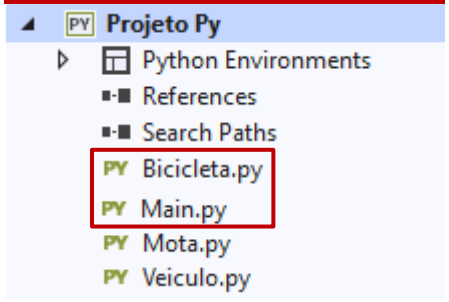
- Cábula:

```
from Veiculo import *

class Mota (Veiculo):
    def __init__(self, cor, marca, peso, cilindrada):
        super().__init__(cor, marca)
        self.__peso = peso
        self.__cilindrada = cilindrada

    def set_peso(self, peso):
        self.__peso = peso

    def get_peso(self):
        return self.__peso

    def receber_combustivel(self):
        print("Estou a receber combustível")

mota = Mota("Vermelho", "Honda", 100, 500)
mota.receber_combustivel()
print("Marca da mota:", mota.get_marca())
```

Herança

Exercício 9 (solução)

Ficheiros do projeto:

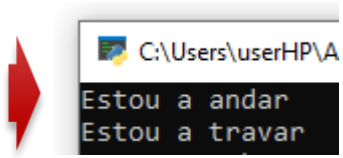


Bicicleta

```
from Veiculo import *

class Bicicleta (Veiculo):
    def __init__(self, cor, marca, tipo, material):
        super().__init__(cor, marca)
        self.__tipo = tipo
        self.__material = material
```

Main

```
from Bicicleta import *

bicicleta = Bicicleta("Azul", "Giant", "BTT", "Fibra de Carbono")
bicicleta.andar()
bicicleta.travar()
```



Herança

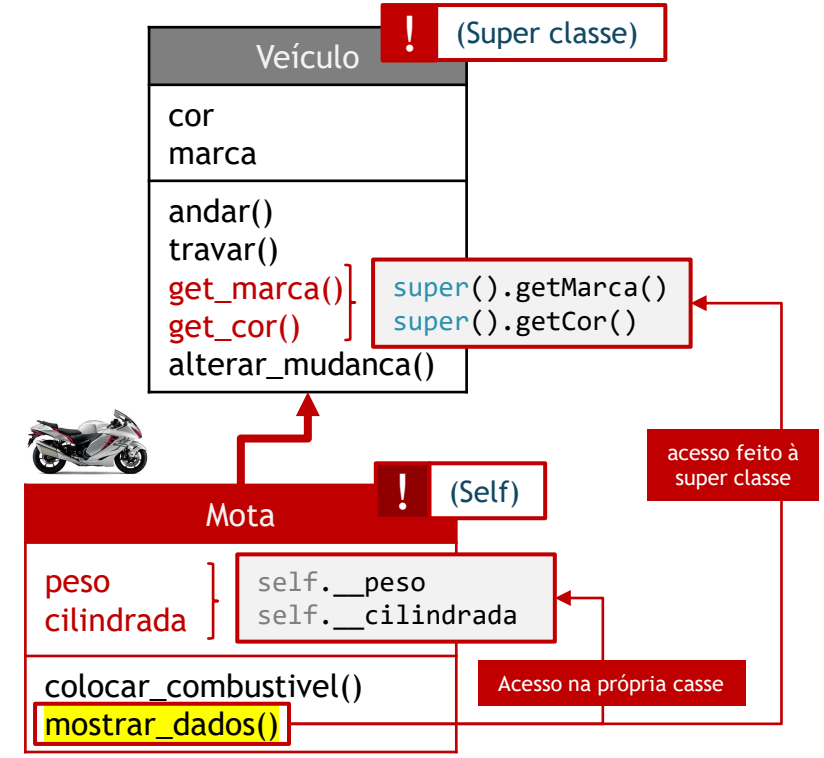
Podemos ainda dentro da classe filha evocar métodos da classe mãe (o inverso não é possível!).

```
from Veiculo import *

class Mota(Veiculo):
    def __init__(self, cor, marca, peso, cilindrada):
        super().__init__(cor, marca)
        self.__peso = peso
        self.__cilindrada = cilindrada

    def set_peso(self, peso):
        self.__peso = peso

    def get_peso(self):
        return self.__peso

    def receber_combustivel(self):
        print("Estou a receber combustível")

    def mostrar_dados(self):
        print(f"Marca:", super().getMarca() )
        print(f"Cor:", super().getCor() )
        print(f"Peso:", self.__peso )
        print(f"Cilindrada:", self.__cilindrada )

mota = Mota("Vermelho", "Honda", 100, 500)
mota.mostrar_dados();
```

```
C:\Users\userHP\Ap
Marca: Honda
Cor: Vermelho
Peso: 100
Cilindrada: 500
```

Elementos que são da classe mãe poderão ser acedidos pela super() classe

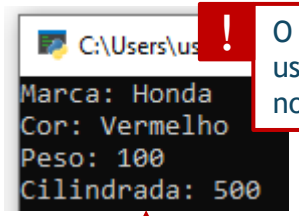
Elementos da classe filha são acedidos pelo self



Herança

Em vez de super() podemos usar o “self”, uma vez que estando os métodos herdados, farão parte do objeto - o resultado será o mesmo, mas não terá o mesmo sentido lógico!

```
from Veiculo import *

class Mota(Veiculo):
    def __init__(self, cor, marca, peso, cilindrada):
        super().__init__(cor, marca)
        self.__peso = peso
        self.__cilindrada = cilindrada

    def set_peso(self, peso):
        self.__peso = peso

    def get_peso(self):
        return self.__peso

    def receber_combustivel(self):
        print("Estou a receber combustível")

    def mostrar_dados(self):
        print(f"Marca:", self.get_marca() )
        print(f"Cor:",   self.get_cor()   )
        print(f"Peso:", self.__peso )
        print(f"Cilindrada:", self.__cilindrada)

mota = Mota("Vermelho", "Honda", 100, 500)
mota.mostrar_dados();
```



! O resultado será o mesmo de usar super().+nome do método no entanto não será tão lógico!

! No entanto no método construtor temos que manter o super()!

Herança

Note que para para conseguirmos aceder a propriedades e métodos da classe mãe, estes precisam de estar como públicos, o que não é o correto a considerar, porque assim o objeto terá sempre acesso (o que nem sempre é desejado).

Como privado

```
class personagem:
    def __init__(self, x, y):
        self.__x = x
        self.__y = y
        self.__em_salto = False

class player(personagem):
    def __init__(self, x, y):
        super().__init__(x, y)

    def saltar(self):
        print( self.__em_salto )

super_mario = player(50, 50)
superMario.saltar()
```

Erro: Propriedades privadas das classes mãe não ficam acessíveis às classes filha!

```
print( self.__emSalto )
```

Exception Thrown

'player' object has no attribute '_player__emSalto'

Como público

```
class personagem:
    def __init__(self, x, y):
        self.__x = x
        self.__y = y
        self.em_salto = False

class player(personagem):
    def __init__(self, x, y):
        super().__init__(x, y)

    def saltar(self):
        print(self.em_salto)

super_mario = player(50, 50)
super_mario.em_salto = True
super_mario.saltar()
```

Acesso em demasia! Publicamente o objeto também dá acesso!

Herança

É neste contexto que entra o modificador de acesso “protected” (protegido) - Todas as propriedades ou métodos que sejam definidas como “protected” são apenas partilhados entre a hierarquia, sendo que o objeto nunca lhes deverá dar acesso públicamente.

Utilizamos o prefixo “_” (apenas 1 underscore) para definir uma propriedade ou método como proteced.

No entanto, ao contrário de outras linguagens de programação, esta funcionalidade nunca foi implementada no Python, trata-se apenas de uma convenção!

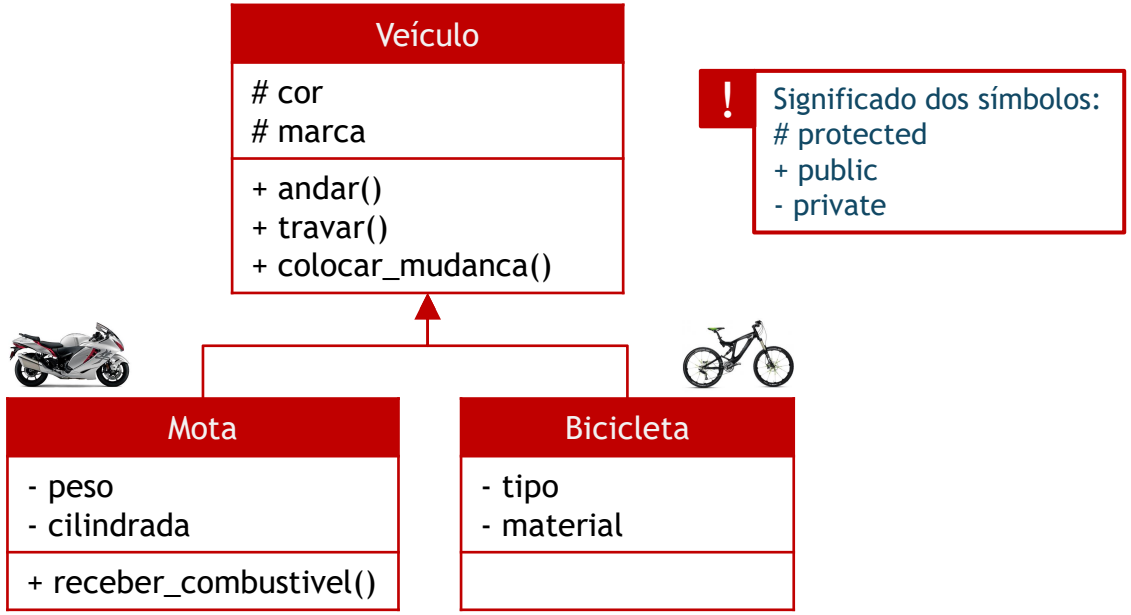
Como protegido

```
class personagem:
    def __init__(self, x, y):
        self.__x = x
        self.__y = y
        self._em_salto = False

class player(personagem):
    def __init__(self, x, y):
        super().__init__(x, y)

    def saltar(self):
        print(self._em_salto)

super_mario = player(50, 50)
print("Estado atual do salto", super_mario._em_salto)
superMario.saltar()
```

! Acesso “inacessível” pelo objeto!

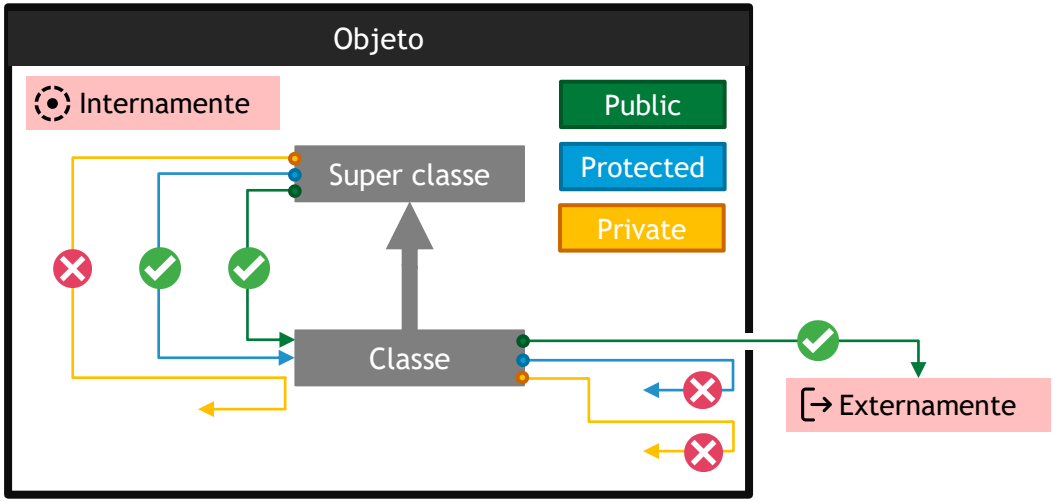
Herança

Curiosidade: Sempre que vir esta representação já saberá que se trata de um modelo de classes UML, que pretende representar por exemplo um Software, API, Framework, etc. :



Encapsulamento – visão geral

Diferença entre Public, Protected e Private:



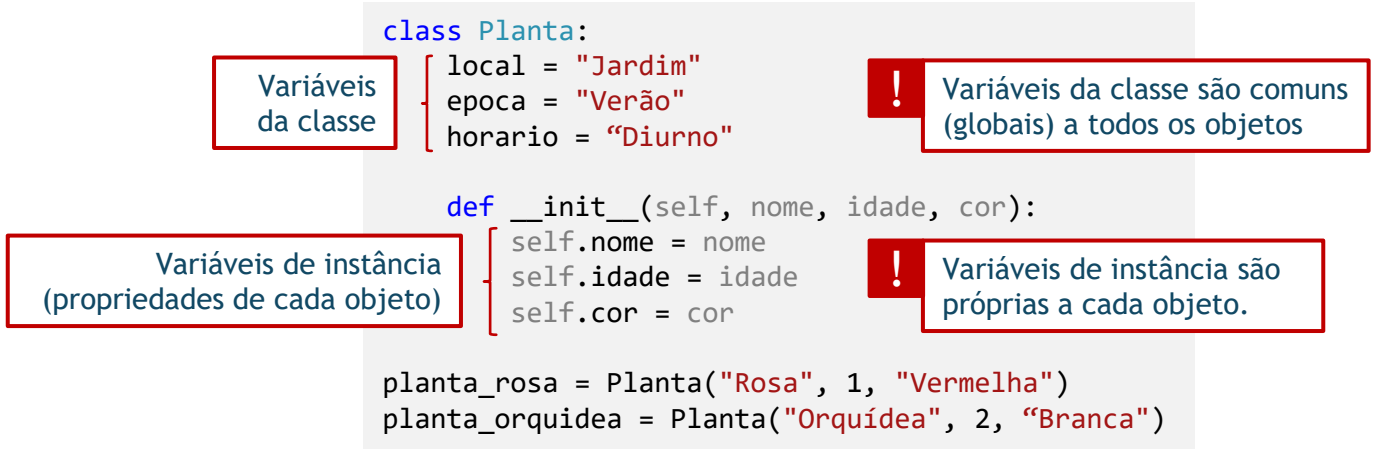
POO

Variáveis de classe e instância

Variáveis de classe e instância

No POO existem 2 tipos de variáveis: Variáveis de classe e variáveis de instância.

As variáveis de classe são também conhecidas por variáveis estáticas. Estas variáveis são criadas apenas uma vez na memória e partilham um acesso comum em todos os objetos. Já no caso das variáveis de instância, estas variáveis são criadas individualmente para cada objeto, ou seja, cada instância da classe possui a sua própria cópia dessas variáveis.

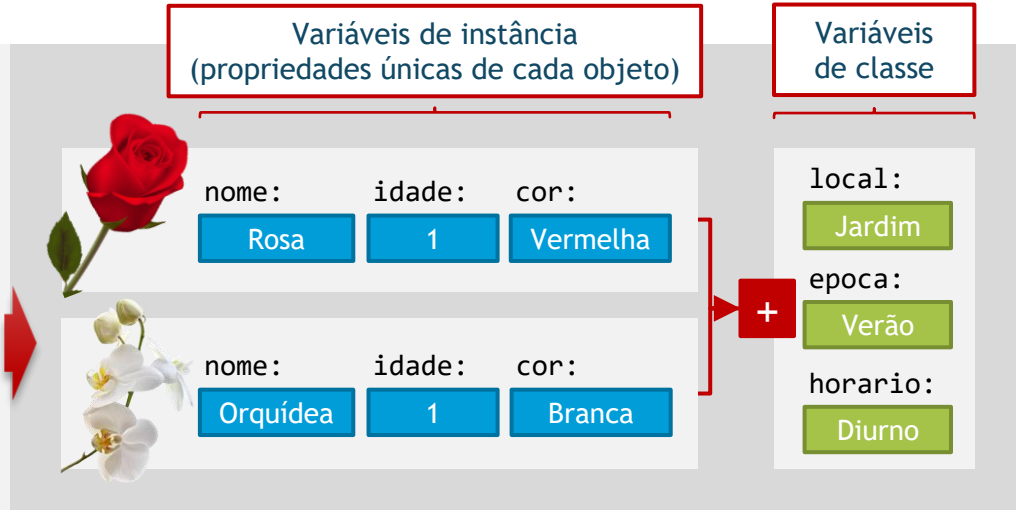


Variáveis de classe e instância

As variáveis de instância são únicas por cada novo objeto criado. As variáveis de classe são partilhadas por todos os objetos.

Com base nas variáveis da classe, no exemplo abaixo, podemos dizer que todas as flores (atuais e futuras) estão localizadas no jardim, na época do ano verão e no horário diurno:

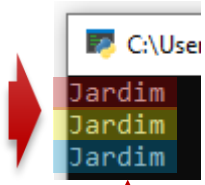
```
class Planta:
    local = "Jardim"
    epoca = "Verão"
    horario = “Diurno"

    def __init__(self, nome, idade, cor):
        self.nome = nome
        self.idade = idade
        self.cor = cor

planta_rosa = Planta("Rosa", 1, "Vermelha")
planta_orquidea = Planta("Orquídea", 2, “Branca")
```



Variáveis de classe e instância

As variáveis de classe podem ser acedidas pelos objetos ou pela própria classe:

```
class Planta:
    local = "Jardim"
    epoca = "Verão"
    horario = "Dia"
    def __init__(self, nome, idade, cor):
        self.nome = nome
        self.idade = idade
        self.cor = cor

planta_rosa = Planta("Rosa", 1, "Vermelha")
planta_orquidea = Planta("Orquídea", 2, "Amarela")

print(planta_rosa.local)
print(planta_orquidea.local)
print(Planta.local)
```



! As variáveis da classe também são acessíveis pelo nome da classe

Variáveis de classe e instância

Para alterar o valor de uma variável de classe, esta alteração deve ser feita somente a partir da classe e nunca do objeto:

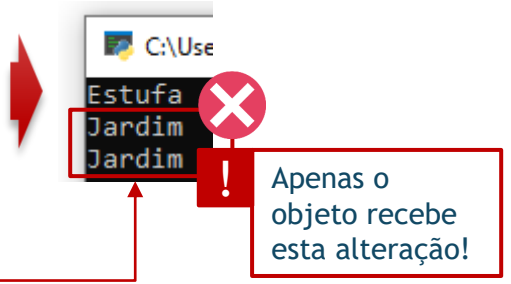
```
class Planta:
    local = "Jardim"
    epoca = "Verão"
    horario = "Dia"

    def __init__(self, nome, idade, cor):
        self.nome = nome
        self.idade = idade
        self.cor = cor

planta_rosa = Planta("Rosa", 1, "Vermelha")
planta_orquidea = Planta("Orquídea", 2, "Amarela")

Planta.local = "Estufa"

print(planta_rosa.local)
print(planta_orquidea.local)
print(Planta.local)
```

Alteração do valor da variável de classe

Variáveis de classe e instância

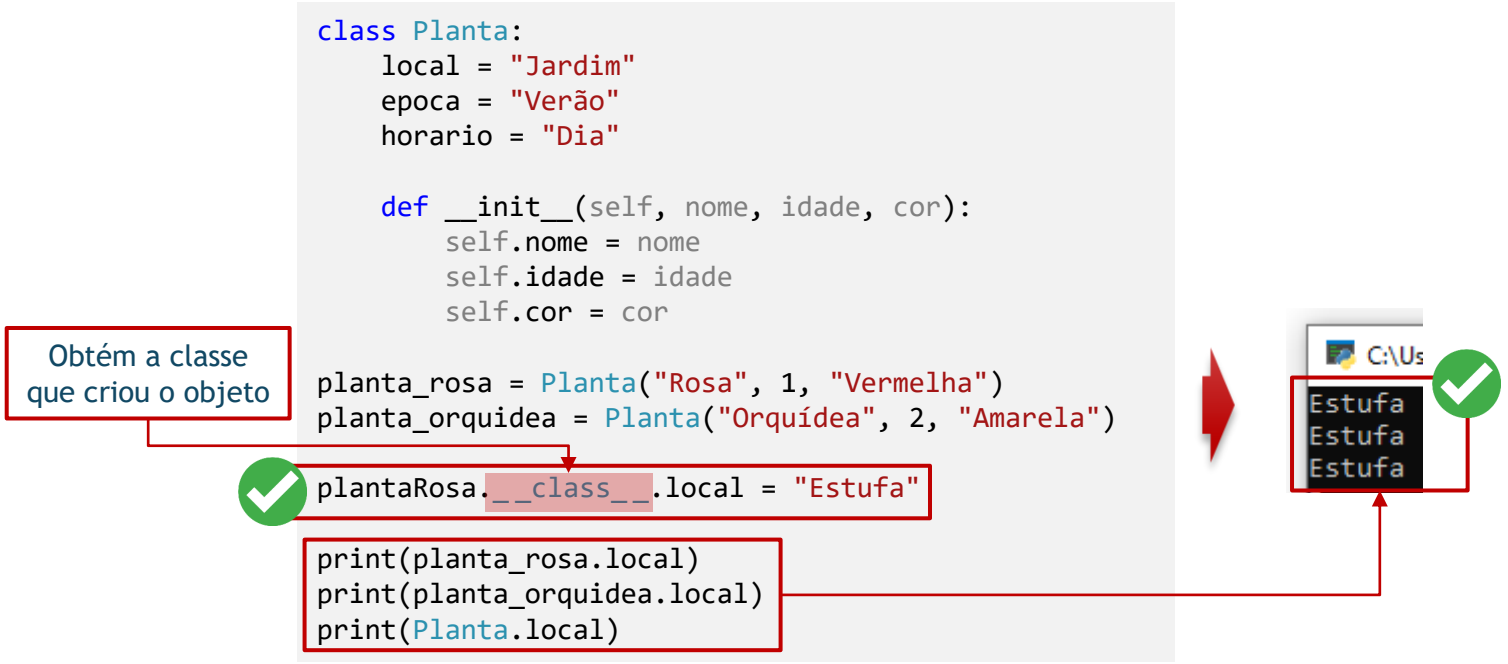
Não é possível alterar o valor de uma variável de classe diretamente pelo objeto isto resulta numa atribuição à superfície do objeto, fazendo com que este perca a referência para aceder à variável da classe.

```
class Planta:
    local = "Jardim"
    epoca = "Verão"
    horario = "Dia"

    def __init__(self, nome, idade, cor):
        self.nome = nome
        self.idade = idade
        self.cor = cor

planta_rosa = Planta("Rosa", 1, "Vermelha")
planta_orquidea = Planta("Orquídea", 2, "Amarela")

planta_rosa.local = "Estufa"

print(planta_rosa.local)
print(planta_orquidea.local)
print(Planta.local)
```



Variáveis de classe e instância

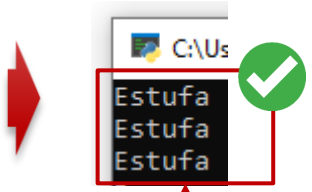
Existe uma alternativa (não recomendado) para alterar dados de variáveis da classe a partir do objeto. Isto implica obter do objeto a classe que o instanciou e seguidamente realizar a alteração diretamente com a pretendida variável da classe:



Variáveis de classe e instância

Para realizar alterações a variáveis de classe dentro da própria classe, é sempre necessário chamar a classe:

```
class Planta:
    local = "Jardim"
    epoca = "Verão"
    horario = "Dia"

    def __init__(self, nome, idade, cor):
        self.nome = nome
        self.idade = idade
        self.cor = cor

    def mudar_local(self, novo_local):
        Planta.local = novo_local

planta_rosa = Planta("Rosa", 1, "Vermelha")
planta_orquidea = Planta("Orquídea", 2, "Amarela")

planta_rosa.mudar_Local("Estufa")

print(planta_rosa.local)
print(planta_orquidea.local)
print(Planta.local)
```



Variáveis de classe e instância

Exercício 10

Crie uma classe que possa representar um computador:

- Cada computador deverá ter uma marca e um sistema operativo.
- Todos os computadores deverão ter um dono em comum (que inicialmente deverá ser visto como “ninguém”).

Seguidamente crie 3 computadores, atribua-lhes o seu nome como o sendo o dono, apresente os dados e o respetivo dono.

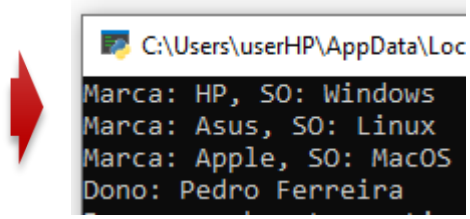
- Cábula:

```
class Planta:
    local = "Jardim"
    epoca = "Verão"
    horario = "Dia"

    def __init__(self, nome, idade, cor):
        self.nome = nome
        self.idade = idade
        self.cor = cor

    def mudar_local(self, novo_local):
        Planta.local = novo_local

planta_rosa = Planta("Rosa", 1, "Vermelha")
planta_orquidea = Planta("Orquídea", 2, "Amarela")

planta_rosa.mudar_local("Estufa")

print(planta_rosa.local)
print(planta_orquidea.local)
print(Planta.local)
```

Variáveis de classe e instância

Exercício 10 (solução)

```
class Computador:
    dono = "Ninguém"

    def __init__ (self, marca, sistema):
        self.marca = marca
        self.sistema = sistema

    def get_dados(self):
        return f"Marca: {self.marca}, SO: {self.sistema}"


computador_win = Computador("HP", "Windows")
computador_lin = Computador("Asus", "Linux")
computador_mac = Computador("Apple", "MacOS")

Computador.dono = "Pedro Ferreira"
print(computador_win.get_dados())
print(computador_lin.get_dados())
print(computador_mac.get_dados())
print(F"Dono: {Computador.dono}")
```



POO

Métodos de classe e instância

Métodos de classe e instância

No POO existem 2 tipos de métodos: Métodos de classe e métodos de instância.

```python
class Planta:
    local = "Jardim"
    epoca = "Verão"
    horario = "Dia"

    def __init__(self, nome, idade, cor):
        self.nome = nome
        self.idade = idade
        self.cor = cor

    @classmethod
    def mudar_local(cls, novo_local):
        cls.local = novo_local

planta_rosa = Planta("Rosa", 1, "Vermelha")
planta_orquidea = Planta("Orquídea", 2, "Amarela")

Planta.mudar_local("Estufa")

print(planta_rosa.local)
print(planta_orquidea.local)
print(Planta.local)
```

Método da instância

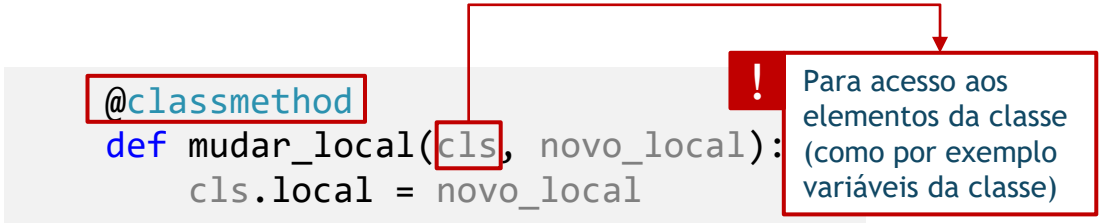
! Métodos da instância são respeitantes a cada objeto individualmente

Método da classe

! Métodos da classe são respeitantes à classe.

Métodos de classe e instância

Os métodos da classe necessitam do declarador “@classmethod”.



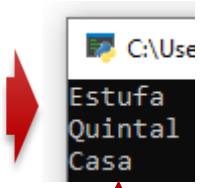
Métodos de classe e instância

Os métodos da classe não têm acesso às propriedades dos objetos (variáveis de instância), apenas têm acesso ás variáveis da classe.

Métodos de classe e instância

Os métodos da classe podem ser diretamente chamados pela classe ou por qualquer objeto por esta instanciado:

```
class Planta:
    local = "Jardim"
    epoca = "Verão"
    horario = "Dia"

    def __init__(self, nome, idade, cor):
        self.nome = nome
        self.idade = idade
        self.cor = cor

    @classmethod
    def mudar_local(cls, novo_local):
        cls.local = novo_local

planta_rosa = Planta("Rosa", 1, "Vermelha")
planta_orquidea = Planta("Orquídea", 2, "Amarela")

Planta.mudar_local("Estufa")
print(Planta.local)
planta_rosa.mudar_local("Quintal")
print(Planta.local)
planta_orquidea.mudar_local("Casa")
print(Planta.local)
```



Métodos de classe e instância

Exercício 11

Crie uma classe que possa representar uma fita-cola.

Cada fita-cola deverá ter o seu comprimento.

Todas as fita-colas terão o mesmo preço (inicialmente 0€).

Crie um método que permita alterar o preço de todas as fita-colas de uma só vez.

Crie 2 objetos fita-cola, atribua-lhes um comprimento (diferente para cada uma) e um preço comum de 2.34€.

Apresente os dados das 2 fita-colas e o seu preço comum.

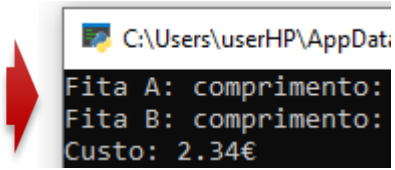
- Cábula:

```
class Planta:
    local = "Jardim"
    epoca = "Verão"
    horario = "Dia"

    def __init__(self, nome, idade, cor):
        self.nome = nome
        self.idade = idade
        self.cor = cor

    @classmethod
    def mudar_local(cls, novo_local):
        cls.local = novo_local

planta_rosa = Planta("Rosa", 1, "Vermelha")
planta_orquidea = Planta("Orquídea", 2, "Amarela")

Planta.mudar_local("Estufa")
print(Planta.local)
planta_rosa.mudar_local(“Quintal")
print(Planta.local)
planta_orquidea.mudar_local(“Casa")
print(Planta.local)
```

Métodos de classe e instância

Exercício 11 (solução)

```
class Fita_Cola:
    preco = 0

    def __init__(self, comprimento):
        self.comprimento = comprimento

    @classmethod
    def alterar_preco(cls, preco):
        cls.preco = preco

    def get_dados(self):
        return f"comprimento: {self.comprimento}"


fita_1 = Fita_Cola(20)
fita_2 = Fita_Cola(30)

Fita_Cola.alterar_preco(2.34)

print(F"Fita-cola 1: {fita_1.getDados()}")
print(F"Fita-cola 2: {fita_2.getDados()}")
print(F"Custo: {Fita_Cola.preco}€")
```

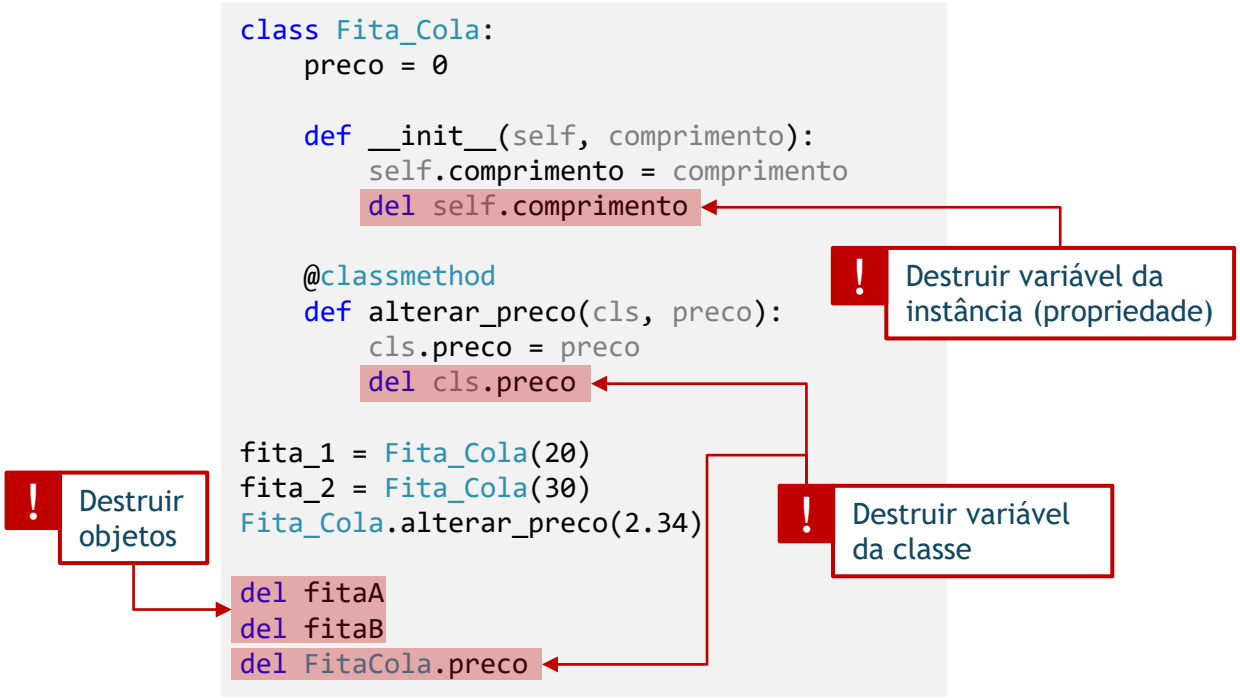


POO

Apagar objetos e variáveis

Apagar objetos e variáveis

Sempre que já não seja precisa uma variável ou objeto e pretendamos libertar memória, podemos usar a instrução “del” para a excluir:



POO

Vantagens e desvantagens

Vantagens e desvantagens

Vantagens:

- Reflete o mundo real, tornando-se semanticamente mais fácil de entender e programar em POO.
- O código passa a estar organizado por grupos (classes), não se misturando com outro código.
- Pode ser programado em paralelo inclusive por vários programadores.
- Reutilização de código: dentro do próprio programa ou inclusive para outros programas.
- Fácil de modificar
- Menos suscetível a falhas
- Cada objeto fica responsável pelas suas características (propriedades) e comportamentos (métodos).

Desvantagens:

- Mais complexo e pesado para o computador do que comparado com a programação estruturada!
- Nem todos os programas podem ou se justificam ser modelados em POO
- A idealização e organização das classes e objetos poderá não coincidir com a visão de outro programador.
- Obriga a que se crie muita documentação para que outros programadores se possam envolver no código.