


# Módulo de Programação Python: Introdução à Linguagem

## Quarto Encontro

### Apresentação de Python: Programação Orientada a Objetos em Python.

No description has been provided for this image

**Objetivo:** Introduzir conceitos de Programação Orientada a Objetos (**POO**). Introduzir o tratamento de exceções como uma aplicação de **POO** aplicado ao tratamento de erros e falhas em tempo de execução.

### Programação Orientada a Objetos

O termo Programação Orientada a Objetos (**POO**) gera algum desconforto ainda entre muitos programadores. De forma geral objetos são algo tangível que podemos sentir, tocar e manipular.


A definição de objeto, em termos de software, não é muito diferente. Objetos não são coisas tangíveis, mas são implementações de modelos de algo que pode fazer certas coisas e fazer com que certas coisas sejam feitas com eles. Formalmente, um objeto é uma coleção de dados e ações associados aos mesmos.

O uso do conceito de objetos em programação envolve uma série de etapas que estão estreitamente relacionadas. De fato, análise, desenho e programação são estágios do desenvolvimento de software, de forma geral. Quando associamos a eles o termo "orientado a objetos" simplesmente estamos especificando qual estilo de desenvolvimento de software está sendo seguido.

A análise orientada a objetos (**AOO**) se refere ao processo de analisar um problema, sistema ou tarefa, que alguém deseja transformar em um aplicativo, e identificar os objetos e interações entre os mesmos. A etapa de análise é sobre "*o que precisa ser feito*".

Já o desenho orientado a objetos (**DOO**) envolve o processo de converter os requisitos, levantados durante a etapa de análise, em um conjunto de especificações para implementação. O designer deve nomear os objetos, definir os comportamentos e especificar formalmente quais objetos podem ativar comportamentos específicos em outros objetos. O estágio de desenho é sobre "*como as coisas devem ser feitas*".

Finalmente, a **POO** define o processo de converter o desenho, já definido, em um programa funcional que faz exatamente o que o cliente solicitou originalmente. Neste curso abordaremos apenas os aspectos relacionados a como executar a etapa de **POO**, utilizando os recursos disponíveis na linguagem **Python 3**

No description has been provided for this image

### Objetos e Classes

Antes de começar temos que deixar dois conceitos fundamentais bem esclarecidos. Como apresentado anteriormente, objetos são conjuntos de dados e ações associadas aos mesmos. Mas na prática temos tipos diferentes de objetos. Quando falamos em telefones celulares, estamos falando de um conjunto de tipos de objetos que podem ser classificados como dispositivos móveis. Entretanto um celular é um objeto diferente de outro celular do mesmo modelo e fabricante. Já um celular e um *tablet* não são apenas objetos diferentes mas tipos de objetos diferentes, ainda que os dois sejam feitos pelo mesmo fabricante. Neste caso estamos falando de classes de objetos.

Qual é a diferença então entre objetos e classes? Classes descrevem objetos, ou seja, elas são como plantas que permitem criar objetos. Você pode ter três celulares do mesmo modelo e fabricante na sua frente. Cada celular é um objeto distinto, mas os três têm os atributos e comportamentos associados a uma classe: a classe que permite "construir" exemplares daquele modelo de celular. Cada celular é um objeto específico, uma instância feita a partir de uma classe. Utilizaremos então, daqui para frente, o termo instanciar para nos referir ao fato de criar um objeto, ao qual também podemos nos referir como instância.

## Um exemplo simples

 No description has been provided for this image

 No description has been provided for this image

## Objetos em Python

Vamos começar fazendo uma apresentação inicial dos recursos, em termos de sintaxes, disponíveis em **Python**, que permite criar programas orientados a objetos. O primeiro passo é então como criar classes.

### Criando Classes em Python

A declaração de uma classe em **Python** é um processo, em princípio, simples. Se utiliza a palavra reservada `class` seguido do nome da classe e de `:` para iniciar o bloco sintático com a implementação da mesma.

```
In [1]: # Criando uma classe
class MinhaPrimeiraClasse :
    pass # implementação do corpo da classe
```

A classe anterior já pode ser utilizada para criar objetos derivados dela, mesmo sem contar ainda com uma definição apropriada da sua implementação. Os objetos ou instâncias de uma classe são criados e, eventualmente, atribuídos a uma variável de forma bastante intuitiva. veja o exemplo a seguir:

```
In [2]: # Criando um instância da classe (objeto)
meuPrimeiroObjeto = MinhaPrimeiraClasse() # chamada ao construtor da classe
# Uma outra instância
outroObjeto = MinhaPrimeiraClasse()
print(meuPrimeiroObjeto)
print(type(meuPrimeiroObjeto))
```

```
<__main__.MinhaPrimeiraClasse object at 0x7fa55882a770>
<class '__main__.MinhaPrimeiraClasse'>
```

Ainda que sem uma implementação formal, as instâncias criadas já podem ser utilizadas.

### Adicionando atributos na classe

Mas, se o conceito de objetos envolve dados, como armazenar informação nos objetos que criamos? Vamos desenvolver uma classe simples para representar ponto no plano cartesiano pelas suas coordenadas x e y.

```
In [3]: # Criamos a classe Ponto sem corpo
class Ponto:
    pass
```

```
In [4]: # Podemos criar duas instâncias (objetos) desta classe
p1 = Ponto()
p2 = Ponto()
```

Usando as instancias da classe podemos definir atributos para cada uma delas da seguinte forma?

```
In [5]: # Agora podemos adicionar atributos às instâncias
p1.x = 1.0
p1.y = 1.0

p2.x = -1.0
p2.y = -1.0
```

Da mesma forma podemos acessar os atributos criados

```
In [6]: # Veja como acessar os atributos de um objeto
print("p1 = (", p1.x, ", ", p1.y, ")")
print("p2 = (", p2.x, ", ", p2.y, ")")
print("p1.__dict__ = ", p1.__dict__)
print("p2.__dict__ = ", p2.__dict__)
print(type(p1.__dict__))
```

```
p1 = ( 1.0 , 1.0 )
p2 = ( -1.0 , -1.0 )
p1.__dict__ = {'x': 1.0, 'y': 1.0}
p2.__dict__ = {'x': -1.0, 'y': -1.0}
<class 'dict'>
```

Repare que, no exemplo anterior, os atributos das instâncias da classe foram definidos após as mesmas serem criadas. Esta não é uma forma comum de declarar atributos de um objeto. Este comportamento peculiar está mais relacionado com a característica de **Python** no tratamento e declaração de variáveis do que com **POO** propriamente.

## Fazendo a classe trabalhar

Agora vamos associar alguma ação que permita modificar de alguma forma os atributos, ou seja o estado de um objeto da classe.

```
In [7]: class Ponto:

    def naOrigem(self): #Método da classe
        self.x = 0.0 #atributos da instância
        self.y = 0.0
```

Os métodos se definem de forma semelhante a funções. Uma diferença fundamental é o fato de que todo método tem, como primeiro parâmetro, `self`. Veja que o parâmetro `self` é utilizado para declarar a variável `x` que, neste caso, referencia um atributo da instância. Este atributo só existe após chamar o método em questão.

O parâmetro `self` representa uma referência ao objeto e, com ajuda dele, é possível declarar e acessar os atributos e métodos de cada objeto.

```
In [8]: # Agora criamos uma instância da nova classe
p1 = Ponto()
# Repare que este objeto ainda não tem atributos
try:
    print("p1 = (", p1.x, ", ", p1.y, ")")
except Exception as e:
    print(e)
print("p1.__dict__ = ", p1.__dict__)
```

```
'Ponto' object has no attribute 'x'
p1.__dict__ = {}
```

Veja que tentamos acessar os atributos `x` e `y` antes de eles existirem. Agora veja o que acontecem se chamamos antes o método `naOrigem` .

```
In [9]: # Agora vamos pedir para colocar o ponto na origem
p1.naOrigem()
try:
    print("p1 = (", p1.x, ", ", p1.y, ")")
except Exception as e:
    print(e)
print("p1.__dict__ = ", p1.__dict__)
```

```
p1 = ( 0.0 , 0.0 )
p1.__dict__ = {'x': 0.0, 'y': 0.0}
```

Novamente os atributos da instância foram declarados após a criarmos o objeto `p1` . Mas desta vez eles foram declarados por uma função definida dentro da classe como um método. A função `naOrigem()` que implementamos na classe `Ponto` , introduz uma funcionalidade na mesma: define que o ponto estará localizado na origem do sistema de coordenadas. Ela tem, basicamente, a mesma sintaxes de uma função em **Python**. Como já comentamos, a principal diferença para uma função simples é que os métodos de uma classe incluem na sua lista de parâmetros e como primeiro parâmetros o `self` .

O parâmetro `self` é simplesmente uma referência ao objeto no qual o método está sendo chamado, ou seja uma autorreferência. É através dele que referenciamos atributos e métodos da própria instância e é exatamente isso que fazemos dentro do método `naOrigem` quando acessamos os atributos `x` e `y` do próprio objeto. Repare que, como os atributos não tinham sido declarados antes, eles estão sendo declarados neste método. Ou seja, quando chamamos o método, se os atributos já existirem eles são atualizados, caso contrario eles são declarados. A referência `self` permite distinguir uma referência a um objeto local, uma variável ou uma função, de um objeto da instância da classe. Reparem que quando chamamos o método do objeto `p` não adicionamos nenhum parâmetro, nem mesmo o `self` . **Python** gerencia isso de forma automática adicionando a referência como primeiro argumento na chamada de qualquer método de um objeto. Veja no exemplo a seguir:

```
In [10]: # Vamos modificar o método
class Ponto:

    # Método da classe ou estático
    def distOrigem(x, y): #Método da classe
        """
        Calcula a distância do ponto (x,y) à origem
        """
        return (x**2 + y**2)**0.5

    def naOrigem(self): #Método da classe
        self.x = 0.0 #atributos da instância
        self.y = 0.0
        # distância não é um atributo, mas uma variável local
```

```
distancia = Ponto.distOrigem(self.x, self.y)
print("Distância da origem = ", distancia)
```

No exemplo anterior criamos um método que não recebe o parâmetro `self`. Neste caso estamos criando um método da classe. No método `naOrigem` declaramos a variável `distancia`, que é uma variável local da função. Veja que usamos também o método `distOrigem`, que calcula a distância de um ponto, dadas as suas coordenadas, à origem do sistema.

```
In [11]: # Agora criamos uma instância da nova classe
p1 = Ponto()
p1.naOrigem()
try:
    print("p1 = (", p1.x, ", ", p1.y, ")", " - distância = ", p1.distancia)
except Exception as e:
    print(e)

try:
    print("p1 = (", p1.x, ", ", p1.y, ")", " - distância = ", Ponto.distOrigem(p1.x,
    print("ou?")
    print("p1 = (", p1.x, ", ", p1.y, ")", " - distância = ", p1.distOrigem(p1.x, p1.
except Exception as e:
    print(e)
```

```
Distância da origem = 0.0
'Ponto' object has no attribute 'distancia'
p1 = ( 0.0 , 0.0 ) - distância = 0.0
ou?
Ponto.distOrigem() takes 2 positional arguments but 3 were given
```

No exemplo anterior, quando tentamos chamar o método da classe desde uma das suas instâncias ele tentou passar o `self` como parâmetro implícito.

Vamos adicionar mais algumas funcionalidades ao nosso modelo de ponto no plano cartesiano, através de novos métodos da classe.

```
In [12]: class Ponto:

    def movePara(self, x = 0, y = 0):
        self.x = x
        self.y = y

    def naOrigem(self):
        self.movePara()

    def distanciaAté(self, outroPonto):
        dist = ((self.x - outroPonto.x)**2 +
                (self.y - outroPonto.y)**2)**0.5
        return dist

    def distOrigem(x, y): #Método da classe
        """
        Calcula a distância do ponto (x,y) à origem
        """
        return (x**2 + y**2)**0.5
```

```
In [13]: # Criando uma instância da nova classe
p1 = Ponto()
# Criando os atributos do objetos na origem do sistema
p1.naOrigem()
```

```
In [14]: # Criando uma nova instância da classe
p2 = Ponto()
```

```
# Criando os atributos do objeto e deslocando um ponto específico
p2.movePara(1.0, 1.0)
```

```
In [15]: # Qual a distância de p2 até p1?
print("A distância de p2 a p1:", p2.distânciaAté(p1))

'''
    O assert é uma verificação em tempo de execução de uma condição qualquer.
    Se a condição não for verdadeira, uma exceção AssertionError acontece
    e o programa pára.
'''

try: # a distância de p1 a p2 deve ser igual à distância de p2 a p1
    assert ( p2.distânciaAté(p1) ==
             p1.distânciaAté(p2) )
    print("O método que calcula a distância funciona!")
except AssertionError:
    print("O método que calcula a distância não funciona!")
```

A distância de p2 a p1: 1.4142135623730951  
O método que calcula a distância funciona!

```
In [16]: # Movendo p1 para novas coordenadas
p1.movePara(4,3)
print("A distância de p1 a p2:", p1.distânciaAté(p2))
print("A distância de p1 a p1:", p1.distânciaAté(p1))
```

A distância de p1 a p2: 3.605551275463989  
A distância de p1 a p1: 0.0

## Inicializando objetos

Até aqui inicializamos os atributos diretamente ou através de métodos específicos das instâncias da classe. Entretanto ainda temos problemas com a garantia de que estes atributos existam e estejam inicializados de forma apropriada no momento que seja necessário utilizar eles.

```
In [17]: # Criamos uma instância da classe Ponto
p1 = Ponto()
# Inicializamos diretamente o atributo x
p1.x = 1.0
# Vamos tentar imprimir as coordenadas de p1
try:
    print("p1 = (", p1.x, ", ", p1.y, ")")
except Exception as err:
    print(err)
```

'Ponto' object has no attribute 'y'

Ou seja, temos sempre que lembrara de inicializar os atributos de uma forma ou de outra. Para evitar esta inconsistência se define em **POO** o conceito de construtor da classe, um método especial que é sempre chamado, automaticamente, no momento que o objeto esta sendo criado e antes de qualquer outro método da classe. Em **Python** se utiliza o método de inicialização `init` com esta finalidade.

```
In [18]: class Ponto:

    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y
        #ou simplesmente
        #self.movePara(x, y)

    def movePara(self, x = 0, y = 0):
        self.x = x
        self.y = y
```

```

def naOrigem(self):
    self.x = 0
    self.y = 0
    #ou simplesmente
    #self.movePara()

def distânciaAté(self, outroPonto):
    dist = ( (self.x - outroPonto.x)**2 +
              (self.y - outroPonto.y)**2 )**0.5
    return dist

def distOrigem(x, y): #Método da classe
    '''
        Calcula a distância do ponto (x,y) à origem
    '''
    return (x**2 + y**2)**0.5

```

```

In [19]: # Criamos novamente uma instância da classe Ponto
p1 = Ponto()
# Inicializamos diretamente o atributo x
p1.x = 1.0
# Vamos tentar imprimir as coordenadas de p1
try:
    print("p1 = (", p1.x, ", ", p1.y, ")")
except Exception as err:
    print(err)

```

```
p1 = ( 1.0 , 0 )
```

Repare que até agora não abordamos o tema de encapsulamento. Voltaremos a este assunto mais para frente.

Nas aulas anteriores comentamos um pouco sobre documentar funções. Vajam um exemplo de documentação da classe que implementamos até aqui.

```

In [20]: class Ponto:
    ''' A Classe Ponto é utilizada para ilustrar a implementação de
        classes em Python e instanciação das mesmas pela criação de
        objetos derivados desta classe. Ela implementa um modelo de
        ponto no plano cartesiano.
    '''

    def __init__(self, x = 0, y = 0):
        ''' Inicialização de um ponto no plano cartesiano. Os valores
            das coordenadas x e y devem ser especificados. Caso os
            valores não sejam definidos o ponto sera criado com
            coordenadas na origem do plano.
        '''
        self.movePara(x, y)

    def movePara(self, x = 0, y = 0):
        ''' Move o ponto para as novas coordenadas (x, y) definidas nos
            parâmetros de entrada. Caso não sejam definidos os valores
            das novas coordenadas, o ponto sera deslocado para a origem.
        '''
        self.x = x
        self.y = y

    def naOrigem(self):
        '''Move o ponto para a origem do sistema de coordenadas cartesianas.'''
        self.movePara()

    def distânciaAté(self, outroPonto):
        ''' Calcula a distância deste ponto até um outro ponto'''

```



```

    dist = ( (self.x - outroPonto.x)**2 +
              (self.y - outroPonto.y)**2 )**0.5
    return dist

def distOrigem(x, y): #Método da classe
    """
        Calcula a distância do ponto (x,y) à origem
    """
    return (x**2 + y**2)**0.5

```

In [21]: `help(Ponto)`

Help on class Ponto in module \_\_main\_\_:

```

class Ponto(builtins.object)
|   Ponto(x=0, y=0)
|
|   A Classe Ponto é utilizada para ilustrar a implementação de
|   classes em Python e instanciação das mesmas pela criação de
|   objetos derivados desta classe. Ela implementa um modelo de
|   ponto no plano cartesiano.
|
|   Methods defined here:
|
|   __init__(self, x=0, y=0)
|       Inicialização de um ponto no plano cartesiano. Os valores
|       das coordenadas x e y devem ser especificados. Caso os
|       valores não sejam definidos o ponto sera criado com
|       coordenadas na origem do plano.
|
|   distOrigem(x, y)
|       Calcula a distância do ponto (x,y) à origem
|
|   distânciaAté(self, outroPonto)
|       Calcula a distância deste ponto até um outro ponto
|
|   movePara(self, x=0, y=0)
|       Move o ponto para as novas coordenadas (x, y) definidas nos
|       parâmetros de entrada. Caso não sejam definidos os valores
|       das novas coordenadas, o ponto sera deslocado para a origem.
|
|   naOrigem(self)
|       Move o ponto para a origem do sistema de coordenadas cartesianas.
|
|   -----
|   Data descriptors defined here:
|
|   __dict__
|       dictionary for instance variables (if defined)
|
|   __weakref__
|       list of weak references to the object (if defined)

```

## Definindo atributos da classe

Os atributos podem ser declarados diretamente no corpo da classe. Mas neste caso eles ganham um escopo diferente: eles passam a ser atributos da classe e não mais das instâncias. Ou seja, o novo atributo, que já existe mesmo antes de ser criada uma instância da classe, referencia uma variável que é compartilhada entre todas as instâncias. Vaja o exemplo seguinte onde modificamos a implementação da classe `Ponto`.

In [22]: `from math import atan2, sin, cos`



```

class Ponto:
    ''' A Classe Ponto é utilizada para ilustrar a implementação de
        classes em Python e instanciação das mesmas pela criação de
        objetos derivados desta classe. Ela implementa um modelo de
        ponto no plano cartesiano.
    '''
    # incluímos um novo atributo na definição do corpo da classe
    contador = 0

    def __init__(self, pCar = None, pPol = None, outroPonto = None):
        '''
            Inicialização de um ponto no plano. Utilizamos coordenadas cartesianas
            e coordenadas polares para representar o ponto em relação à origem do
            sistema de coordenadas. As coordenadas do ponto podem ser fornecidas
            como coordenadas cartesianas, polares ou ainda como outro ponto. Caso os
            valores não sejam definidos o ponto sera criado com
            coordenadas na origem do sistema de coordenadas.
            Cada ponto criado ganha um id único construído com base
            no atributo contador. O atributo também permite contabilizar
            quantas instâncias da classe já foram criadas.
            O construtor pode ser chamado de três formas diferentes:
            1. Utilizando uma dupla com as coordenadas x,y do ponto
            em coordenadas cartesianas:
            p1 = Ponto(pCar = (1.0,2.3))
            2. Utilizando uma dupla com as coordenadas rho,theta do ponto
            em coordenadas polares:
            p1 = Ponto(pPol = (1,3.14))
            3. Utilizando outro ponto:
            p2 = Ponto(outroPonto = p1)
            4. Ou ainda sem parâmetros:
        '''
        if pCar:
            x, y = pCar
            self.__x = x
            self.__y = y
            self.__rho, self.__theta = Ponto.cart2pol()
        elif pPol:
            rho, theta = pPol
            self.__rho = rho
            self.__theta = theta
            self.__x, self.__y = Ponto.pol2cart()
        elif outroPonto and type(outroPonto) == 'Ponto':
            self.__x = outroPonto.__x
            self.__y = outroPonto.__y
            self.__rho = outroPonto.__rho
            self.__theta = outroPonto.__theta
        else:
            self.moveXYPara()

        # Acessando o atributo da classe
        Ponto.contador += 1 # Podemos referencia assim
        self.__id = "Ponto-" + str(self.contador) # ou assim

    # métodos da classe
    def cart2pol(x, y):
        rho = (x**2 + y**2)**0.5
        theta = atan2(y, x)
        return rho, theta

    def pol2cart(rho, theta):
        x = rho * cos(theta)
        y = rho * sin(theta)
        return x, y

```

```

def clonarPonto(self):
    return Ponto((self.__x, self.__y))

def moveXYPara(self, x = 0, y = 0):
    ''' Move o ponto para as novas coordenadas (x, y) definidas nos
        parâmetros de entrada. Caso não sejam definidos os valores
        das novas coordenadas, o ponto sera deslocado para a origem.
    '''
    self.__x = x
    self.__y = y
    self.__rho, self.__theta = self.cart2pol(self.__x, self.__y)

def moveXPara(self, x = 0):
    '''

    '''
    self.__x = x
    self.__rho, self.__theta = self.cart2pol(self.__x, self.__y)

def moveYPara(self, y = 0):
    '''

    '''
    self.__y = y
    self.__rho, self.__theta = self.cart2pol(self.__x, self.__y)

def naOrigem(self):
    '''Move o ponto para a origem do sistema de coordenadas cartesianas.'''
    self.moveXYPara()

def naOrigem(self):
    'Move o ponto para a origem.'
    self.movePara()

def moveRhoThetaPara(self, rho = 0, theta = 0):
    '''

    '''
    self.__rho = rho
    self.__theta = theta
    self.__x, self.__y = self.pol2cart(self.__rho, self.__theta)

def moveRhoPara(self, rho = 0):
    '''

    '''
    self.__rho = rho
    self.__x, self.__y = self.pol2cart(self.__rho, self.__theta)

def moveThetaPara(self, theta = 0):
    '''

    '''
    self.__theta = theta
    self.__x, self.__y = self.pol2cart(self.__rho, self.__theta)

def distanciaAté(self, outroPonto):
    ' Calcula a distância deste ponto até um outro ponto'
    dist = ( (self.__x - outroPonto.__x)**2 +
              (self.__y - outroPonto.__y)**2 )**0.5
    return dist

```

Esta classe implementa várias funcionalidades e, pela primeira vez trata o controverso tema de encapsulamento em **Python**.

Em **Python** podemos identificar o atributos como públicos, protegidos e privados.

- Atributos cujo identificador começa com letra são intrinsecamente públicos. Ou seja, podem ser acessados diretamente desde as instâncias da classe.
- Atributos cujo identificador começa com um subscrito (*underline* `_`) são atributos marcados como privados pelo desenvolvedor. Na prática eles continuam sendo públicos mas, o desenvolvedor deixou uma marca recomendando que eles sejam tratados como privados. Ainda podem ser acessados diretamente desde as instâncias da classe mas se recomenda que sejam acessados desde os métodos apropriados. Podemos chamar eles como protegidos pelo desenvolvedor ou fracamente privados, mesmo que esta classificação não se encaixe na tradicionalmente utilizada em **POO**.
- Atributos cujo identificador começa com dois subscritos (*underlines* `__`) são atributos marcados como fortemente privados pelo desenvolvedor. Na prática eles não podem mais ser acessados diretamente, mesmo que ainda seja possível acessar eles utilizando mecanismos mais "obscuros" da linguagem. Podemos chamar eles como fortemente privados, mesmo que esta classificação não se encaixe na tradicionalmente utilizada em **POO**.

Veja os exemplos

```
In [23]: class Ponto3D:
    def __init__(self, x = 0, y = 0, z = 0):
        self.x = x      # publico
        self._y = y     # fracamente privado
        self.__z = z    # fortemente privado

    def setX(self, x):
        self.x = x

    def setY(self, y):
        self._y = y

    def setZ(self, z):
        self.__z = z

    def getX(self):
        return self.x

    def getY(self):
        return self._y

    def getZ(self):
        return self.__z
```

```
In [24]: p1 = Ponto3D(1,2,3)
#Podemos acessar o atributo público diretamente
print("p1.x = ", p1.x)
p1.x = 2
print("p1.x = ", p1.x)
# ou pelo método getX e setX
print("p1.getX() = ", p1.getX())
p1.setX(10)
print("p1.getX() = ", p1.getX())
```

```
p1.x = 1
p1.x = 2
p1.getX() = 2
p1.getX() = 10
```

```
In [25]: # Podemos acessar o atributo fracamente privado diretamente
print("p1._y = ", p1._y)
p1._y = 2
```

```

print("p1._y = ", p1._y)
# ou pelo método getY e setY
print("p1.getY() = ", p1.getY())
p1.setY(10)
print("p1.getY() = ", p1.getY())

```

```

p1._y = 2
p1._y = 2
p1.getY() = 2
p1.getY() = 10

```

```

In [26]: # Podemos tentar acessar o atributo fortemente privado diretamente
p1 = Ponto3D(1,2,3)
try:
    print("p1.__z = ", p1.__z)
except Exception as e:
    print(e)

```

'Ponto3D' object has no attribute '\_\_z'

```

In [27]: #Isto não da erro, mas funciona como a gente espera?
print("p1.__dict__ = ", p1.__dict__)
p1.__z = 2
print("p1.__z = ", p1.__z)
print("p1.__dict__ = ", p1.__dict__)

```

```

p1.__dict__ = {'x': 1, '_y': 2, '_Ponto3D__z': 3}
p1.__z = 2
p1.__dict__ = {'x': 1, '_y': 2, '_Ponto3D__z': 3, '__z': 2}

```

```

In [28]: # Veja quando usamos os métodos getZ e setZ
print("p1.getZ() = ", p1.getZ())
p1.setZ(10)
print("p1.getZ() = ", p1.getZ())

print("p1.__dict__ = ", p1.__dict__)

```

```

p1.getZ() = 3
p1.getZ() = 10
p1.__dict__ = {'x': 1, '_y': 2, '_Ponto3D__z': 10, '__z': 2}

```

```

In [29]: # Podemos acessar o atributo fortemente privado diretamente
p1 = Ponto3D(1,2,3)
print("p1._Ponto3D__z = ", p1._Ponto3D__z)
p1._Ponto3D__z = 2
print("p1._Ponto3D__z = ", p1._Ponto3D__z)
print("p1.getZ() = ", p1.getZ())

```

```

p1._Ponto3D__z = 3
p1._Ponto3D__z = 2
p1.getZ() = 2

```

O exemplo a seguir foi implementado no módulo anterior.

```

In [30]: class Carro:
    def __init__(self, cor, marca, modelo, ano):
        self.cor = cor
        self.marca = marca
        self.modelo = modelo
        self.ano = ano

    def acelerar(self, velocidade):
        print("Acelerando o carro a ", velocidade, " km/h")

    def frear(self):
        print("Freando o carro")

```

```
In [31]: civic = Carro("preto", "Honda", "Civic", 2019)
print("civic.cor = ", civic.cor)
print("civic.marca = ", civic.marca)
print("civic.modelo = ", civic.modelo)
print("civic.ano = ", civic.ano)

civic.acelerar(100)
civic.frear()
```

```
civic.cor = preto
civic.marca = Honda
civic.modelo = Civic
civic.ano = 2019
Acelerando o carro a 100 km/h
Freando o carro
```

Então, podemos concluir que o mecanismo de encapsulamento em **Python** funciona mais como uma pática de progamação baseada nos princípios de **POO** do que como propriamente uma restrição forte.

Uma alternativa para contornar esta restrição é utilizar decoradores ( `decorators` ), que são um recurso simples mais muito poderoso da linguagem.

Antes de passarmos para decoradores, devemos lembrar que funções em python são objetos!

Isso significa que uma função pode ser:

- repassado como argumento
- usado em expressões
- retornado como valores de outras funções

Assim como números inteiros ou string!

Os decoradores são um pouco como papel de presente. São funções que pegam outra função, adicionam alguma funcionalidade e retornam a nova função "decorada".

```
In [32]: # Decorador
def funçãoDecoradora(funçãoEntrada):
    def novaFunção(arg): # arg é o argumento da função decorada
        print("Adiciono algo dntes da execução da função")
        funçãoEntrada(arg)
        print("E/ou adiciono algo depois da execução da função")
    return novaFunção

def falaOi(nome):
    print("Oi", nome)

falaOi = funçãoDecoradora(falaOi)

falaOi("João")
```

```
Adiciono algo dntes da execução da função
Oi João
E/ou adiciono algo depois da execução da função
```

Agora vamos introduzir um tipo especial de *decorator* utilizado para tratar métodos *seter* e *geter* de classes, visando facilitar a implementação do encapsulamento de atributos.

```
In [33]: class Ponto3D:
    def __init__(self, x = 0, y = 0, z = 0):
        self.__x = x # todos os atributos declarados como fortemente privados
        self.__y = y # mas poderiam ser fracamente privados ou mesmo públicos
        self.__z = z
```

```

#decorador que faz uma função se comportar como um atributo (propriedade)
@property
def x(self):
    return self.__x

#decorador que faz uma função se comportar como um atributo (propriedade)
@x.setter
def x(self, x):
    self.__x = x

@property
def y(self):
    return self.__y

@y.setter
def y(self, y):
    self.__y = y

@property
def z(self):
    return self.__z

@z.setter
def z(self, z):
    self.__z = z

```

Agora podemos usar os métodos como atributos. Veja como

```

In [34]: p1 = Ponto3D(1,2,3)
print("p1.x = ", p1.x)
p1.x = 2
print("p1.x = ", p1.x)
p1.y = 10
print("p1.y = ", p1.y)
p1.z = 20
print("p1.z = ", p1.z)
print("p1.__dict__ = ", p1.__dict__)

```

```

p1.x = 1
p1.x = 2
p1.y = 10
p1.z = 20
p1.__dict__ = {'_Ponto3D__x': 2, '_Ponto3D__y': 10, '_Ponto3D__z': 20}

```

```

In [35]: class Carro:
    def __init__(self, cor, marca, modelo, ano):
        self.__cor = cor
        self.__marca = marca
        self.__modelo = modelo
        self.__ano = ano
        self.__velocidade = 0

    @property
    def cor(self):
        return self.__cor

    @cor.setter
    def cor(self, cor):
        self.__cor = cor

    @property
    def marca(self):
        return self.__marca

```

```

@marca.setter
def marca(self, marca):
    self.__marca = marca

@property
def modelo(self):
    return self.__modelo

@modelo.setter
def modelo(self, modelo):
    self.__modelo = modelo

@property
def ano(self):
    return self.__ano

@ano.setter
def ano(self, ano):
    self.__ano = ano

@property
def velocidade(self):
    return self.__velocidade

def acelerar(self, velocidade):
    self.__velocidade = velocidade
    print("Acelerando o carro a ", velocidade, " km/h")

def frear(self):
    self.__velocidade = 0
    print("Freando o carro")

```

## Herança

Herança em **Python** se implementa de forma relativamente simples. Vamos começar por herança simples

```

In [36]: class Ponto2D:
    def __init__(self, x = 0, y = 0):
        self.__x = x      # todos os atributos declarados como fortemente privados
        self.__y = y      # mas poderiam ser fracamente privados ou mesmo públicos

    @property
    def x(self):
        return self.__x

    @x.setter
    def x(self, x):
        self.__x = x

    @property
    def y(self):
        return self.__y

    @y.setter
    def y(self, y):
        self.__y = y

    def movePara(self, x = 0, y = 0):
        ''' Move o ponto para as novas coordenadas (x, y) definidas nos
            parâmetros de entrada. Caso não sejam definidos os valores
            das novas coordenadas, o ponto sera deslocado para a origem.
            ...
        '''
        self.__x = x

```



```

self.__y = y

def naOrigem(self):
    '''Move o ponto para a origem do sistema de coordenadas cartesianas.'''
    self.movePara()

def distânciaAté(self, outroPonto):
    ''' Calcula a distância deste ponto até um outro ponto'''
    dist = ( (self.__x - outroPonto.__x)**2 +
              (self.__y - outroPonto.__y)**2 )**0.5
    return dist

def distOrigem(x, y): #Método da classe
    '''
        Calcula a distância do ponto (x,y) à origem
    '''
    return (x**2 + y**2)**0.5

```

Podemos criara classe Ponto3D como uma classe que estende Ponto2D

```

In [37]: class Ponto3D(Ponto2D):
def __init__(self, x = 0, y = 0, z = 0):
    #super().__init__(x, y)
    #ou
    Ponto2D.__init__(self, x, y)
    self.__z = z

@property
def z(self):
    return self.__z

@z.setter
def z(self, z):
    self.__z = z

def movePara(self, x = 0, y = 0, z = 0):
    ''' Move o ponto para as novas coordenadas (x, y, y) definidas nos
        parâmetros de entrada. Caso não sejam definidos os valores
        das novas coordenadas, o ponto sera deslocado para a origem.
    '''
    #super().movePara(x, y)
    #ou
    Ponto2D.movePara(self, x, y)
    self.__z = z

def distânciaAté(self, outroPonto):
    ''' Calcula a distância deste ponto até um outro ponto'''
    dist = ( (self.__x - outroPonto.__x)**2 +
              (self.__y - outroPonto.__y)**2 +
              (self.__z - outroPonto.__z)**2 )**0.5
    return dist

def distOrigem(x, y, z): #Método da classe
    '''
        Calcula a distância do ponto (x,y) à origem
    '''
    return (x**2 + y**2 + z**2)**0.5

```

Com a nova classe podemos criar instâncias e acessar os métodos implementados nela e aqueles que herda da super-classe

```

In [38]: P1 = Ponto3D(1,2,3)
print("P1.x = ", P1.x)
print("P1.y = ", P1.y)

```

```

print("P1.z = ", P1.z)
print("P1.__dict__ = ", P1.__dict__)
P1.movePara(10,20,30)
print("P1.x = ", P1.x)
print("P1.y = ", P1.y)
print("P1.z = ", P1.z)
print("P1.__dict__ = ", P1.__dict__)
P1.naOrigem()
print("P1.x = ", P1.x)
print("P1.y = ", P1.y)
print("P1.z = ", P1.z)
print("P1.__dict__ = ", P1.__dict__)

p2 = Ponto2D(1,2)
print("p2.x = ", p2.x)
print("p2.y = ", p2.y)
print("p2.__dict__ = ", p2.__dict__)

```

```

P1.x = 1
P1.y = 2
P1.z = 3
P1.__dict__ = {'_Ponto2D__x': 1, '_Ponto2D__y': 2, '_Ponto3D__z': 3}
P1.x = 10
P1.y = 20
P1.z = 30
P1.__dict__ = {'_Ponto2D__x': 10, '_Ponto2D__y': 20, '_Ponto3D__z': 30}
P1.x = 0
P1.y = 0
P1.z = 0
P1.__dict__ = {'_Ponto2D__x': 0, '_Ponto2D__y': 0, '_Ponto3D__z': 0}
p2.x = 1
p2.y = 2
p2.__dict__ = {'_Ponto2D__x': 1, '_Ponto2D__y': 2}

```

**Python** fornece alguns recursos para pesquisar a relação entre objetos e classes. `issubclass()` `isinstance()`

```

In [39]: print(issubclass(Ponto2D, Ponto3D))
print(issubclass(Ponto3D, Ponto2D))

```

```

False
True

```

```

In [40]: print(isinstance(p2, Ponto2D))
print(isinstance(p2, Ponto3D))
print(isinstance(P1, Ponto2D))
print(isinstance(P1, Ponto3D))

```

```

True
False
True
True

```

Outro recurso importante implementado em **Python** é a herança múltipla.

```

In [41]: class superClasseA:

    def __init__(self, varA):
        self.__varA = varA
        self.__estado = "A"

    def falaOi(self):
        print("Oi de A!: " + self.__estado)

class subClasseB(superClasseA):
    def __init__(self, varA, varB):

```

```

    super().__init__(varA)
    self.__varB = varB
    self.__estado = "B"

    def falaOi(self):
        print("Oi de B!: " + self.__estado)

class subClasseC(superClasseA):

    def __init__(self, varA, varC):
        super().__init__(varA)
        self.__varC = varC
        self.__estado = "C"

    def falaOi(self):
        print("Oi de C!: " + self.__estado)

class subClasseD(subClasseB, subClasseC):

    def __init__(self, varA, varB, varC, varD):
        super().__init__(varA, varB)
        self.__varD = varD
        self.__estado = "D"

    def falaOi(self):
        print("Oi de D!: " + self.__estado)

```

Veja como funciona a herança neste exemplo que ilustra o **MRO** (*Method Resolution Order*)

```
In [42]: objA = superClasseA("A")
        objA.falaOi()
```

Oi de A!: A

```
In [43]: objB = subClasseB("A", "B")
        objB.falaOi()
```

Oi de B!: B

```
In [44]: objC = subClasseC("A", "C")
        objC.falaOi()
```

Oi de C!: C

```
In [45]: #Uma versão um pouco diferente de da superClasseC
        class superClasseC_(superClasseA):
```

```

            def __init__(self, varA, varC):
                super().__init__(varA)
                self.__varC = varC
                self.__estado = "C"

```

```
In [46]: objC_ = superClasseC_("A", "C")
        objC_.falaOi()
        objC_.__dict__
```

Oi de A!: A

```
Out[46]: {'_superClasseA__varA': 'A',
         '_superClasseA__estado': 'A',
         '_superClasseC__varC': 'C',
         '_superClasseC__estado': 'C'}
```

```
In [47]: # Variações para testar MRO
        class superClasseA:

            def __init__(self, varA):
```

```

        self.__varA = varA
        self.__estado = "A"

    def falaOi(self):
        print("Oi de A!: " + self.__estado)

class subClasseB(superClasseA):
    def __init__(self, varA, varB):
        super().__init__(varA)
        self.__varB = varB
        self.__estado = "B"

    def falaOi(self):
        print("Oi de B!: " + self.__estado)

class subClasseC(superClasseA):

    def __init__(self, varA, varC):
        super().__init__(varA)
        self.__varC = varC
        self.__estado = "C"

    def falaOi(self):
        print("Oi de C!: " + self.__estado)

class subClasseD(subClasseB, subClasseC):

    def __init__(self, varA, varB, varC, varD):
        super().__init__(varA, varB)
        self.__varD = varD
        self.__estado = "D"

    def falaOi(self):
        print("Oi de D!: " + self.__estado)

```

## Polimorfismo

Um dos exemplos que vimos no módulo anterior para ilustrar o **polimorfismo** foi o da classe objeto geométrico. Vejamos como fica este exemplo implementado em **Python**

```

In [48]: class ObjetoGeométricos:
    def __init__(self):
        self.__nome = "Objeto Geométrico"

    def calculaArea(self):
        pass

    def calculaPerímetro(self):
        pass

    def __str__(self):
        return self.__nome

```

```

In [49]: class Triângulo(ObjetoGeométricos):
    def __init__(self, base, altura):
        super().__init__()
        self.__base = base
        self.__altura = altura
        self.__nome = "Triângulo"

    def calculaArea(self):
        return self.__base * self.__altura / 2

```

```

def calculaPerímetro(self):
    return 3 * self.__base

def __str__(self):
    return self.__nome + " Area: " + str(self.calculaArea()) + " Perímetro: " + s

```

In [50]:

```

class Retângulo(ObjetoGeométricos):
    def __init__(self, base, altura):
        super().__init__()
        self.__base = base
        self.__altura = altura
        self.__nome = "Retângulo"

    def calculaArea(self):
        return self.__base * self.__altura

    def calculaPerímetro(self):
        return 2 * self.__base + 2 * self.__altura

    def __str__(self):
        return self.__nome + " Area: " + str(self.calculaArea()) + " Perímetro: " + s

```

O polimorfismo de classes se da em casos como o ilustrado a seguir

In [51]:

```

objG1 = Triângulo(10, 5)
objG2 = Retângulo(10, 5)
objG3 = Triângulo(2, 6)
objG4 = Retângulo(2, 6)
listaObjetos = [objG1, objG2, objG3, objG4]
def areaTotal(listaObjetos):
    area = 0
    for obj in listaObjetos:
        area += obj.calculaArea()
    return area
for obj in listaObjetos:
    print(obj)

print("Area total: ", areaTotal(listaObjetos))

```

```

Triângulo Area: 25.0 Perímetro: 30
Retângulo Area: 50 Perímetro: 30
Triângulo Area: 6.0 Perímetro: 6
Retângulo Area: 12 Perímetro: 16
Area total: 93.0

```

Mas o que acontece se definimos a seguinte classe?

In [52]:

```

class Circunferência:
    def __init__(self, raio):
        self.__raio = raio
        self.__nome = "Circunferência"

    def calculaArea(self):
        return 3.14 * self.__raio**2

    def calculaPerímetro(self):
        return 2 * 3.14 * self.__raio

    def __str__(self):
        return self.__nome + " Area: " + str(self.calculaArea()) + " Perímetro: " + s

```

In [53]:

```

objG5 = Circunferência(10)
objG6 = Circunferência(2)

```

```

listaObjetos.append(objG5)
listaObjetos.append(objG6)
for obj in listaObjetos:
    print(obj)
print("Area total: ", areaTotal(listaObjetos))

```

```

Triângulo Area: 25.0 Perímetro: 30
Retângulo Area: 50 Perímetro: 30
Triângulo Area: 6.0 Perímetro: 6
Retângulo Area: 12 Perímetro: 16
Circunferência Area: 314.0 Perímetro: 62.800000000000004
Circunferência Area: 12.56 Perímetro: 12.56
Area total: 419.56

```

Podemos concluir que o polimorfismo de classes não é necessário em **Python**. Ou seja, não é necessário uma relação hierárquica entre as classes para podermos utilizar um objeto de uma classe no lugar de outro. Basta que elas implementem os mesmos métodos, mesmo que com funcionalidades diferentes. Já constatamos isto com objetos de outras classes até aqui.

Entretanto o mecanismo de classes abstratas continua sendo de grande utilidade, já que, com ele, conseguimos forçar as classes filhas a implementar os métodos desejados.

```

In [54]: from abc import ABC, abstractmethod

class ClasseAbstrata(ABC):

    @abstractmethod
    def __init__(self, atributo):
        self._atributo = atributo

    @abstractmethod
    def metodoAbstrato(self):
        pass

class ClasseConcreta(ClasseAbstrata):

    def __init__(self, atributo, atributo2):
        super().__init__(atributo)
        self._atributo2 = atributo2

    def metodoAbstrato(self):
        print("Implementação do método abstrato")

    def metodoConcreto(self):
        print("Implementação do método concreto")

```

Vejamos como isto funciona na classe `ObjetoGeométrico`

```

In [55]: objG1 = ObjetoGeométricos()
objG1 = Triângulo(10, 5)
print(objG1)

```

```

Triângulo Area: 25.0 Perímetro: 30

```

```

In [56]: class ObjetoGeométricos(ABC):

    @abstractmethod
    def __init__(self):
        self.__nome = "Objeto Geométrico"

    @abstractmethod
    def calculaArea(self):
        pass

```

```

@abstractmethod
def calculaPerímetro(self):
    pass

@abstractmethod
def __str__(self):
    return self.__nome

```

```

In [57]: class Triângulo(ObjetoGeométricos):
def __init__(self, base, altura):
    super().__init__()
    self.__base = base
    self.__altura = altura
    self.__nome = "Triângulo"

def calculaArea(self):
    return self.__base * self.__altura / 2

def calculaPerímetro(self):
    return 3 * self.__base

def __str__(self):
    return self.__nome + " Area: " + str(self.calculaArea()) + " Perímetro: " + s

```

```

In [58]: #objG1 = ObjetoGeométricos()
objG1 = Triângulo(10, 5)
print(objG1)

```

Triângulo Area: 25.0 Perímetro: 30

## Métodos especiais

Métodos com função específica.

```

In [59]: #Dunder methods: Métodos especiais
class classeEncap:

    def __init__(self, atributoPublico, atributoProtegido, atributoPrivado):
        self.atributoPublico = atributoPublico
        self._atributoProtegido = atributoProtegido # convenção
        self.__atributoPrivado = atributoPrivado # definido como privado

    def __metodoPrivado(self):
        print("Método privado")

    def metodoPublico(self):
        print("Método público")

    @property
    def atributoPrivado(self):
        return self.__atributoPrivado

    @atributoPrivado.setter
    def atributoPrivado(self, valor):
        self.__atributoPrivado = valor

    @atributoPrivado.deleter
    def atributoPrivado(self):
        del self.__atributoPrivado

    @property
    def atributoProtegido(self):
        return self._atributoProtegido

```



```

@atributoProtegido.setter
def setAtributoProtegido(self, valor):
    self._atributoProtegido = valor

@atributoProtegido.deleter
def atributoProtegido(self):
    del self._atributoProtegido

def __str__(self):
    return "Atributo público: " + str(self.atributoPublico) + \
        "\nAtributo protegido: " + str(self._atributoProtegido) + \
        "\nAtributo privado: " + str(self.__atributoPrivado)

def __repr__(self) :
    return f"classeEncap({self.atributoPublico}, {self._atributoProtegido}, {self.__atributoPrivado})"

def __eq__(self, outro):
    return self.atributoPublico == outro.atributoPublico and \
        self._atributoProtegido == outro._atributoProtegido and \
        self.__atributoPrivado == outro.__atributoPrivado

def __add__(self, outro):
    return classeEncap(self.atributoPublico + outro.atributoPublico,
                        self._atributoProtegido + outro._atributoProtegido,
                        self.__atributoPrivado + outro.__atributoPrivado)

```

```

In [60]: obj = classeEncap(1,2,3)
print(obj)
obj2 = eval(repr(obj))
print(obj2)
print(obj is obj2)
print(obj == obj2)
obj3 = obj + obj2
print(obj3)

```

```

Atributo público: 1
Atributo protegido: 2
Atributo privado: 3
Atributo público: 1
Atributo protegido: 2
Atributo privado: 3
False
True
Atributo público: 2
Atributo protegido: 4
Atributo privado: 6

```