



# **PMR3201 Computação para Automação**

## **Aula de Laboratório 3**

Programação Orientada a Objetos: Herança

---

Newton Maruyama  
Thiago de Castro Martins  
Marcos S. G. Tsuzuki  
1 de maio de 2017

PMR-EPUSP

1. Composição de objetos: de volta a classe Circle
2. Herança
3. Hierarquia de polígonos
4. Para você fazer

## **Composição de objetos: de volta a classe Circle**

---

- A definição de uma classe pode utilizar objetos de outra classe.
- Suponha por exemplo que deseja-se representar as coordenadas do centro do círculo numa classe denominada **Point**

```
class Point():  
    def __init__(self, xValue, yValue):  
        self.x = float(xValue)  
        self.y = float(yValue)  
  
    def printxy(self):          # imprime as variaveis internas de Point  
        print('cordenada x =', self.x)  
        print('cordenada y =', self.y)
```

- A classe **Circle** pode ser definida da seguinte forma:

```
class Circle():
    numero = 0
    def __init__( self, name = 'circle', x = 0, y = 0, radius = 0.0 ):
        self.name = name
        self.coord = Point(x,y)      # cria um objeto da classe point
        self.radius = float( radius )
        Circle.numero = Circle.numero + 1

    def area( self ):
        return math.pi * self.radius ** 2
```

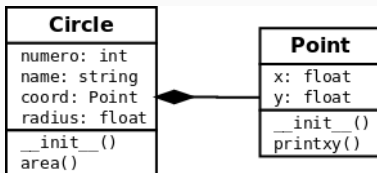
- A variável interna coord é um objeto da classe **Point**.
- Note a presença da variável numero dentro da classe mas sem a palavra reservada **def**.
- Quando um objeto é criado a variável é incrementada através do comando:

```
Circle.numero = Circle.numero + 1
```

- A variável numero é uma variável comum a todos os objetos da classe **Circle**. Todos os objetos conseguem modifica-la.

## Composição de objetos: representação

- ▶ Na terminologia de programação orientada a objetos essa associação entre objetos é denominada composição.
- ▶ Usualmente utiliza-se uma linguagem de modelagem gráfica denominada UML (Unified Modeling Language) para descrever arquiteturas orientadas a objeto.
- ▶ A linguagem provê vários tipos de diagramas para representação estática e dinâmica.
- ▶ Por exemplo, a relação entre a classe **Circle** e a classe **Point** poderia ser representada pelo seguinte diagrama de classes.




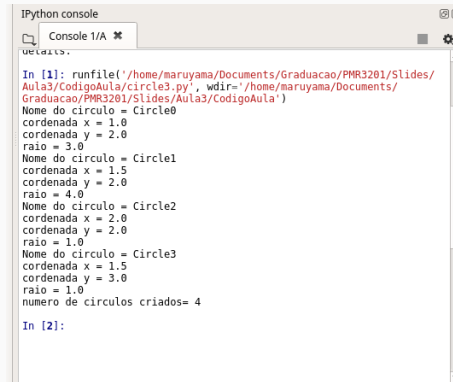
- ▶ O arquivo que contém a definição das classes **Point** e **Circle** além do programa principal que utiliza tais classes se encontra no arquivo `circle3.py`.
- ▶ A seguir temos a listagem do programa principal:

```
def main():
    lista_de_circulos=[]
    # lista aonde sera armazenado os objetos do tipo circulo
    # Lista com parametros que definem circulos
    # x,y,radius
    parametros_do_circulo = [[1.0,2.0,3.0],[1.5,2.0,4.0],
                             [2.0,2.0,1.0],[1.5,3.0,1.0]]
    numero_de_circulos = len(parametros_do_circulo)
    for k in range(numero_de_circulos):
        # nome do circulo gerado como 'Circle'+str(k)
        a = Circle('Circle'+str(k),parametros_do_circulo[k][0],
                  parametros_do_circulo[k][1], parametros_do_circulo[k][2])
        lista_de_circulos.append(a) # insere novo circulo na lista
    # checa o conteudo de cada objeto do tipo circulo contido na lista
    for k in range(numero_de_circulos):
        print('Nome do circulo =',lista_de_circulos[k].name)
        lista_de_circulos[k].coord.printxy(); # funcao da classe Point
        print('raio =',lista_de_circulos[k].radius)
    print('numero de circulos criados=',Circle.numero)
if __name__ == "__main__": main()
```

- ▶ Note que para imprimir o valor das coordenadas do centro do círculo utiliza-se o seguinte comando:

```
lista_de_circulos[k].coord.printxy();
```

- ▶ Carregue o arquivo na IDE Spyder  e verifique o seu funcionamento.
- ▶ A figura abaixo apresenta a saída do programa no console da IDE Spyder:



```
IPython console
Console 1/A ✖
details.
In [1]: runfile('/home/maruyama/Documents/Graduacao/PMR3201/Slides/
Aula3/CodigoAula/circle3.py', wdir='/home/maruyama/Documents/
Graduacao/PMR3201/Slides/Aula3/CodigoAula')
Nome do circulo = Circle0
cordenada x = 1.0
cordenada y = 2.0
raio = 3.0
Nome do circulo = Circle1
cordenada x = 1.5
cordenada y = 2.0
raio = 4.0
Nome do circulo = Circle2
cordenada x = 2.0
cordenada y = 2.0
raio = 1.0
Nome do circulo = Circle3
cordenada x = 1.5
cordenada y = 3.0
raio = 1.0
numero de circulos criados= 4

In [2]:
```

- ▶ Note que a variável `Circle.numero` apresenta o valor 4 que é o número de objetos criados.
- ▶ Cada objeto da classe **Circle** incrementa o valor dessa variável.



## Herança

---

- ▶ Um dos mecanismos mais importantes de Programação Orientada a Objetos é denominado herança.
- ▶ Uma classe pode ser derivada de uma outra classe.
- ▶ Por exemplo, se a classe B for derivada da classe A diz-se que A é a superclasse e B é a subclasse.
- ▶ A classe B "herda" os atributos (variáveis) e métodos (funções) da classe A.
- ▶ Um dos intuitos do mecanismo de herança é a possibilidade de se reutilizar código.
- ▶ Ao invés de se escrever o código de uma classe a partir do "zero" pode-se utilizar a definição de uma classe existente e acrescentar novos atributos e métodos.

- Por exemplo, pode-se derivar a classe **Circle** a partir da classe **Point**

```
class Point():
    def __init__( self, xValue, yValue):
        self.x = float(xValue)
        self.y = float(yValue)

    def printxy(self):          # imprime as variaveis internas de Point
        print('cordenada x =',self.x)
        print('cordenada y =',self.y)

class Circle(Point):
    numero = 0
    def __init__( self, name = 'circle',x = 0, y = 0, radius = 0.0 ):
        self.name = name
        Point.__init__(self, x, y)
        self.radius = float( radius )
        Circle.numero = Circle.numero + 1

    def area( self ):
        return math.pi * self.radius ** 2
```

- Note que a definição da classe `Circle` agora faz referencia à classe **Point**:

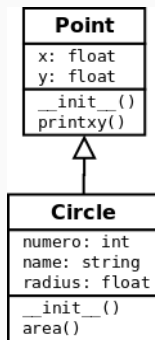
```
class Circle(Point):
```

- Dentro do construtor `__init__` da classe **Circle** deve ser feito a execução do construtor da classe **Point** como observado abaixo:

```
Point.__init__(self, x, y)
```

- Como a classe **Circle** é derivada de **Point** as variáveis `x`, `y` e o método `printxy()` passam a fazer parte da classe **Circle**.

- Na linguagem UML o diagrama de classes representando a relação de herança entre a classe **Circle** e a classe **Point** pode ser representada como a seguir.



- O arquivo que contém a definição das classes **Point** e **Circle** além do programa principal que utiliza tais classes se encontra no arquivo circle4.py.
- A seguir apresenta-se a listagem do programa principal:

```
def main():
    lista_de_circulos=[] # lista aonde sera armazenado os objetos do tipo circulo
    # Lista com parametros que definem circulos
    # x,y,radius
    parametros_do_circulo = [[1.0,2.0,3.0],[1.5,2.0,4.0],[2.0,2.0,1.0],[1.5,3.0,1.0]]
    numero_de_circulos = len(parametros_do_circulo)
    for k in range(numero_de_circulos):
        # nome do circulo e gerado como 'Circle'+str(k)
        a = Circle('Circle'+str(k),parametros_do_circulo[k][0],
            parametros_do_circulo[k][1], parametros_do_circulo[k][2])
        lista_de_circulos.append(a) # insere novo circulo na lista
    # checa o conteudo de cada objeto do tipo circulo contido na lista
    for k in range(numero_de_circulos):
        print('Nome do circulo =',lista_de_circulos[k].name)

        lista_de_circulos[k].printxy(); # funcao da classe Point

        print('raio =',lista_de_circulos[k].radius)
    print('numero de circulos criados=',Circle.numero)
if __name__ == "__main__": main()
```

- ▶ Carregue o arquivo na IDE Spyder  e verifique o seu funcionamento.
- ▶ A figura abaixo apresenta a saída do programa no console da IDE Spyder:



```
IPython console
Console 1/A ✕

In [3]: runfile('/home/maruyama/Documents/Graduacao/PMR3201/Slides/
Aula3/CodigoAula/circle4.py', wdir='/home/maruyama/Documents/
Graduacao/PMR3201/Slides/Aula3/CodigoAula')
Nome do circulo = Circle0
cordenada x = 1.0
cordenada y = 2.0
raio = 3.0
Nome do circulo = Circle1
cordenada x = 1.5
cordenada y = 2.0
raio = 4.0
Nome do circulo = Circle2
cordenada x = 2.0
cordenada y = 2.0
raio = 1.0
Nome do circulo = Circle3
cordenada x = 1.5
cordenada y = 3.0
raio = 1.0
numero de circulos criados= 4

In [4]:
```

- ▶ Novamente a variável `Circle.numero` apresenta o valor 4 que é o número de objetos criados.
- ▶ A chamada ao método `printxy()` é feito da seguinte forma:

```
lista_de_circulos[k].printxy();
```

- ▶ Note que é como se `printxy()` pertencesse à classe **Circle**.

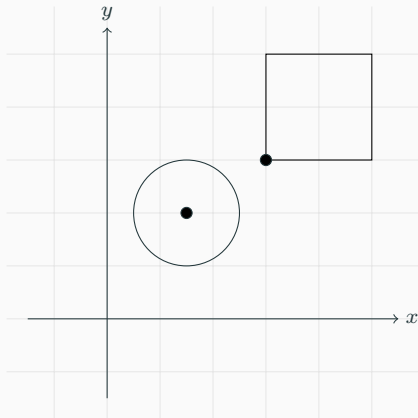
## **Hierarquia de polígonos**

---



## Polígonos: círculos, quadrados

- Deseja-se representar círculos e quadrados



- ▶ No próximo exemplo será estabelecido uma superclasse denominada **Polygon**.
- ▶ Duas classes representando polígonos, **Circle** e **Square**, serão definidas como subclasses de **Polygon**.
- ▶ A ancoragem do quadrado em relação ao sistema de coordenadas é dado pelas coordenadas do ponto que define o canto inferior esquerdo. Somente são admitidos quadrados cuja base está paralela ao eixo x.
- ▶ A classe **Point** nesse caso será utilizada como parte da classe **Polygon**

# Classe Polygon e Point

- Note que a variável numero está definida na classe **Polygon**.
- Observe também a implementação da função area().

```
class Point():
    def __init__( self, xValue, yValue):
        self.x = float(xValue)
        self.y = float(yValue)

    def printxy(self):          # imprime as variaveis internas de Point
        print('cordenada x =',self.x)
        print('cordenada y =',self.y)

class Polygon:
    numero = 0
    def __init__(self,name,x,y):
        self.name=name
        self.coord=Point(x,y) # ponto que ancora o poligono

    def area(self):
        print('Necessario implementacao especifica de area()')
        return(0.0)
```

- ▶ A classe **Circle** é agora uma subclasse de **Polygon**
- ▶ O método `printcontent()` foi definido para imprimir o conteúdo do objeto.
- ▶ Para cada objeto da classe **Circle** criado a variável `Polygon.numero` é incrementada.

```
class Circle(Polygon):
    def __init__(self, name, x, y, radius):
        Polygon.__init__(self, name, x, y)
        self.radius = float(radius)
        Polygon.numero = Polygon.numero + 1

    def area( self ):
        return math.pi * self.radius ** 2

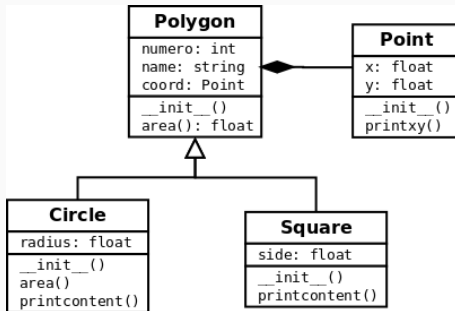
    def printcontent(self):
        print('name =', self.name)
        self.coord.printxy()
        print('radius =', self.radius)
```

- ▶ A classe **Square** também é uma subclasse da classe **Polygon**.
- ▶ Note que não existe a definição de um método `area()`.

```
class Square(Polygon):
    def __init__(self, name='square', x=0, y=0, side=0):
        Polygon.__init__(self, name, x, y)
        self.side = float(side)
        Polygon.numero = Polygon.numero + 1

    def printcontent(self):
        print('name =', self.name)
        self.coord.printxy()
        print('side =', self.side)
```

- Na linguagem UML representaríamos o diagrama de classes como ilustrado a seguir.



- ▶ O programa em questão se encontra no arquivo `polygono.py`.
- ▶ A seguir temos a listagem do programa principal (Parte 1):

```
def main():
    lista_de_poligonos=[] # lista aonde sera armazenado os objetos do tipo Polygon
    # Lista com parametros que definem os poligonos
    parametros_dos_poligonos = [['circle', 'circ0', 1.0, 2.0, 3.0], ['circle', 'circ1', 1.5, 2.0, 4.0],
    ['square', 'square0', 2.0, 2.0, 1.0]]

    numero_de_poligonos = len(parametros_dos_poligonos)


    for k in range(numero_de_poligonos):
        #
        if parametros_dos_poligonos[k][0]=='circle':
            name=parametros_dos_poligonos[k][1]
            x=parametros_dos_poligonos[k][2]
            y=parametros_dos_poligonos[k][3]
            radius=parametros_dos_poligonos[k][4]
            a=Circle(name,x,y,radius)
            lista_de_poligonos.append(a) # insere novo circulo na lista
        elif parametros_dos_poligonos[k][0]=='square':
            name=parametros_dos_poligonos[k][1]
            x=parametros_dos_poligonos[k][2]
            y=parametros_dos_poligonos[k][3]
            side=parametros_dos_poligonos[k][4]
            a=Square(name,x,y,side)
            lista_de_poligonos.append(a) # insere novo square na lista
        else:
            print('Tipo de poligono nao identificado')
```

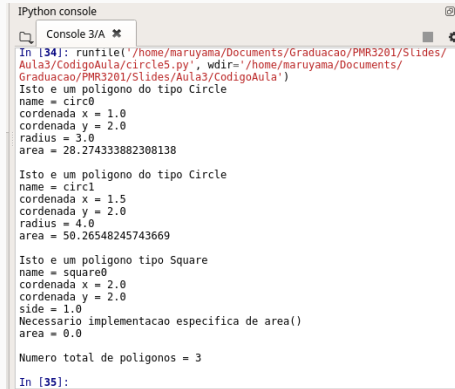
- A seguir temos a listagem do programa principal (Parte 2):

```
numero_de_poligonos_na_lista=len(lista_de_poligonos)
for k in range(numero_de_poligonos_na_lista):
    if type(lista_de_poligonos[k]) is Circle:
        print('Isto e um poligono do tipo Circle')
    else:
        print('Isto e um poligono tipo Square')
        lista_de_poligonos[k].printcontent()
        print('area =', lista_de_poligonos[k].area())
        print()
print('Numero total de poligonos =', Polygon.numero)

if __name__ == "__main__": main()
```



- ▶ Carregue o arquivo na IDE Spyder  e verifique o seu funcionamento.
- ▶ A figura abaixo apresenta a saída do programa no console da IDE Spyder:



```
IPython console
Console 3/A ✖
In [34]: runtile('/home/maruyama/Documents/Graduacao/PMR3201/Slides/
Aula3/CodigoAula/circle5.py', wdir='/home/maruyama/Documents/
Graduacao/PMR3201/Slides/Aula3/CodigoAula')
Isto e um poligono do tipo Circle
name = circ0
cordenada x = 1.0
cordenada y = 2.0
radius = 3.0
area = 28.274333882308138

Isto e um poligono do tipo Circle
name = circ1
cordenada x = 1.5
cordenada y = 2.0
radius = 4.0
area = 50.26548245743669

Isto e um poligono tipo Square
name = square0
cordenada x = 2.0
cordenada y = 2.0
side = 1.0
Necessario implementacao especifica de area()
area = 0.0

Numero total de poligonos = 3

In [35]:
```

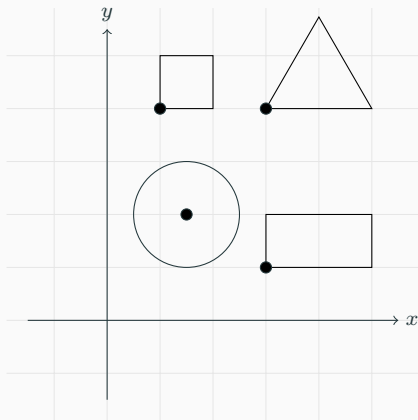
- ▶ Novamente a variável `Polygon.numero` apresenta o valor 3 que é o número de objetos criados da classe **Polygon**.
- ▶ Como a classe **Circle** possui um método `area()` definido este se superpõe ao método de mesmo nome na superclasse **Polygon**.
- ▶ Por outro lado, a classe **Square** não possui um método `area()` definido em sua própria classe por isso a execução do método equivalente na superclasse.
- ▶ Note também a utilização da função `type()` para checar qual objeto está associado a uma variável.

**Para você fazer**

---

## Polígonos: círculos, quadrados, retângulos e triângulos

- Deseja-se representar vários tipos de polígonos através de uma hierarquia de classes na linguagem Python.



## Aumentando os tipos de polígonos

- ▶ Deseja-se agora acrescentar os polígonos da classe **Rectangle** e da classe **Triangle**.
- ▶ A classe **Rectangle** deve ser derivada da classe **Square**. O retângulo sempre fica paralelo ao eixo x. O ponto de ancoragem se refere ao canto inferior esquerdo. A base do retângulo sempre fica paralela ao eixo x.
- ▶ A classe **Triangle** admite apenas triângulos equiláteros com a base paralela ao eixo x. O ponto de ancoragem se refere ao canto esquerdo da base do triângulo.
- ▶ Utilize o arquivo Polygon1.py.

- O sistema a ser implementado pode ser representado pelo seguinte diagrama de classes:

