PMR3201 Computação para Automação
Aula de Laboratório 2

Programação Orientada a Objetos

Newton Maruyama
Thiago de Castro Martins
Marcos S. G. Tsuzuki
Rafael Traldi Moura

1 de abril de 2019

PMR-EPUSP

Conteúdo

Abstração de dados

Tipos de dados abstratos - ADT

- Usualmente os tipos de dados simples da linguagem não são suficientes para a representação de problemas.
- Dessa forma, partindo de tipos de dados simples, as bibliotecas de estruturas de dados (listas, pilhas, árvore binárias, etc.) foram desenvolvidas nas diversas linguagens de programação.
- Posteriormente, verificou-se a conveniência da utilização das estruturas de dados representadas por um conjunto de operações abstratas que transformam o estado das estruturas de dados
- Essa representação é denominada tipos abstratos de dados.
- Nessa representação os dados e operações associadas estão interconectados.

Fila: definição

- Por exemplo o ADT Fila tem um comportamento dinâmico do tipo FIFO (First-In First-Out), ou seja, o primeiro a chegar é o primeiro a sair.
- Uma definição para uma Fila poderia ser como a seguir:

```
structure Fila (of ItemType)
interface
CriaFila → Fila
InsereFila(Fila,ItemType) → Fila
RetiraFila(Fila) → Fila, ItemType
FilaVazia(Fila) → Boolean
FilaCheia(Fila) → Boolean
d = Fila(of ItemType)
D = {Fila,ItemType,Boolean}
F = {CriaFila,InsereFila,RetiraFila,FilaVazia,FilaCheia}
```

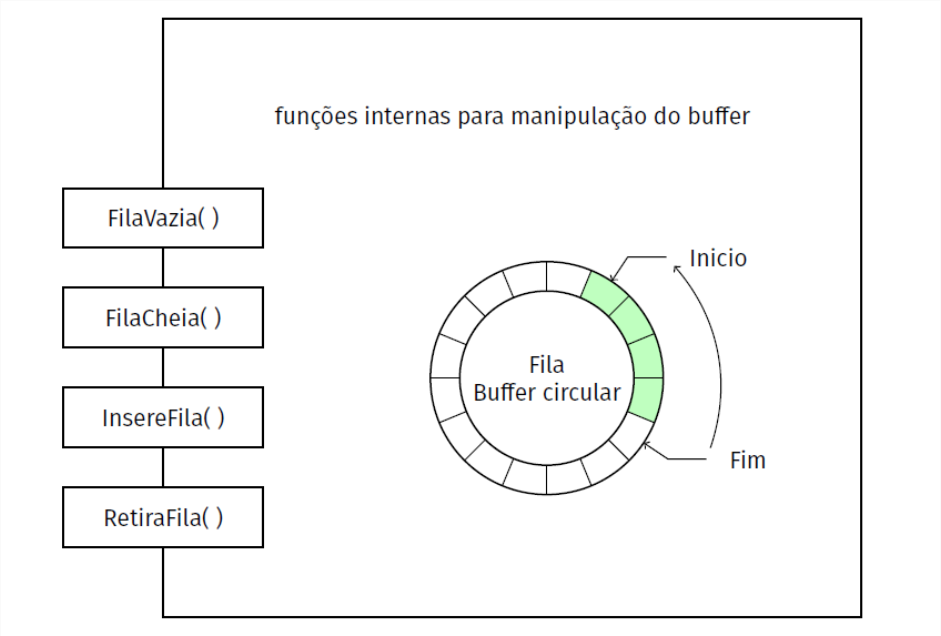
Fila: algumas operações

Seja a seguinte sequencia de operações abstratas:

1. `CriaFila`$() \rightarrow T = ()$
2. `FilaVazia`$() \rightarrow True$
3. `InsereFila`$(T, \texttt{a}) \rightarrow T = (\texttt{a})$
4. `InsereFila`$(T, \texttt{d}) \rightarrow T = (\texttt{a,d})$
5. `InsereFila`$(T, \texttt{b}) \rightarrow T = (\texttt{a,d,b})$
6. `RetiraFila`$(T) \rightarrow T = (\texttt{d,b}),\ a$
7. `RetiraFila`$(T) \rightarrow T = (\texttt{b}),\ d$

Fila: implementação

- A implementação da Fila compreende a transformação de especificações abstratas em especificações concretas utilizando uma linguagem de programação.
- Uma possível idealização deste objeto está ilustrado na Figura abaixo.
- A utilização da Fila se faz através de chamadas de funções que representam as operações da Fila. Essas operações definem uma interface.
- Essas funções se utilizam de funções internas que manipulam a estrutura de dados concreta, por exemplo, um *buffer* circular, que pode ser implementado como um *array*.
- Uma possível função `CriaFila()` é responsável por criar esse objeto.

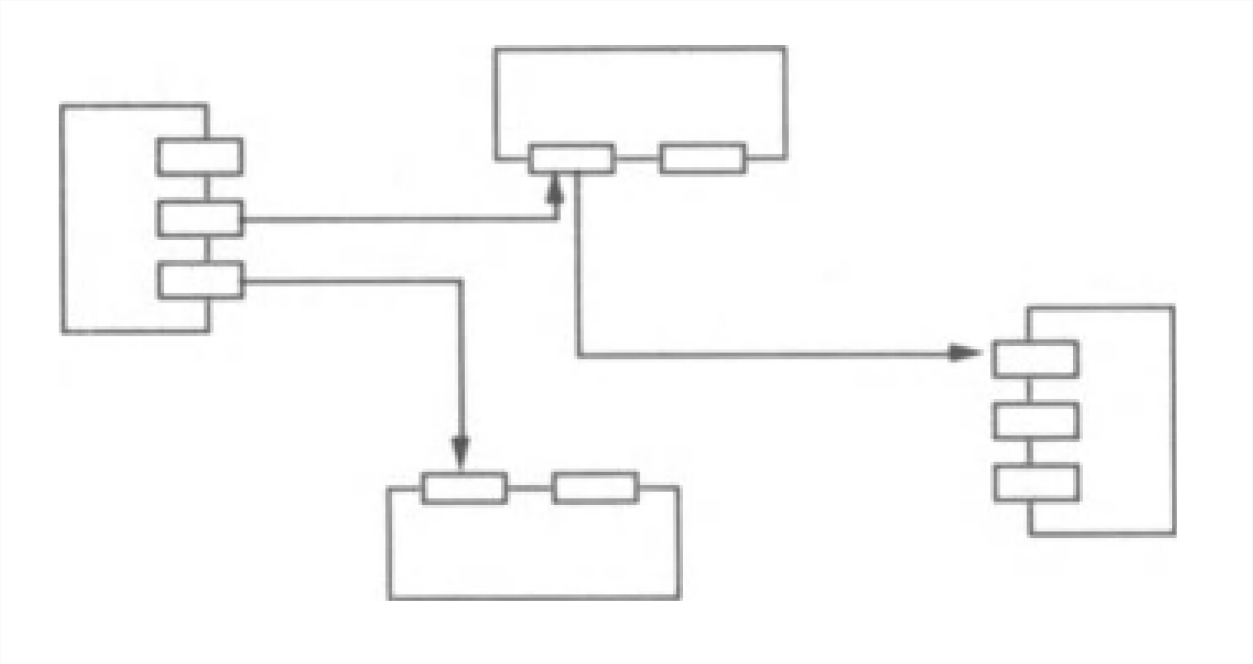


Linguagens orientadas a objetos

- A concepção de tipos abstratos de dados é um dos pontos de partida para a concepção das linguagens orientadas a objetos.
- Nessas linguagens o usuário constrói os seus proprios tipos de dados que são denominados classes.
- Uma classe contêm ao mesmo tempo estruturas de dados e funções (operações) que são indissociáveis.

Arquitetura de sistemas

- A utilização de objetos leva a um tipo de arquitetura de sistema aonde vários objetos se comunicam entre si através das operações que definem suas interfaces.



Tipos de dados na linguagem Python

Variáveis em Python

- Na linguagem Python a definição do tipo da variável é realizada implicitamente.

```
i=1
j=10
x=3.14
y=6.5
```

- Na linguagem C, por exemplo, devemos atribuir explicitamente a cada símbolo um tipo de variável:

```
int i, j;
float x,y;
```

- Uma das características marcantes de Python é a possibilidade de mudança de tipo em tempo de execução:

```
x=3.14
x='a'
```

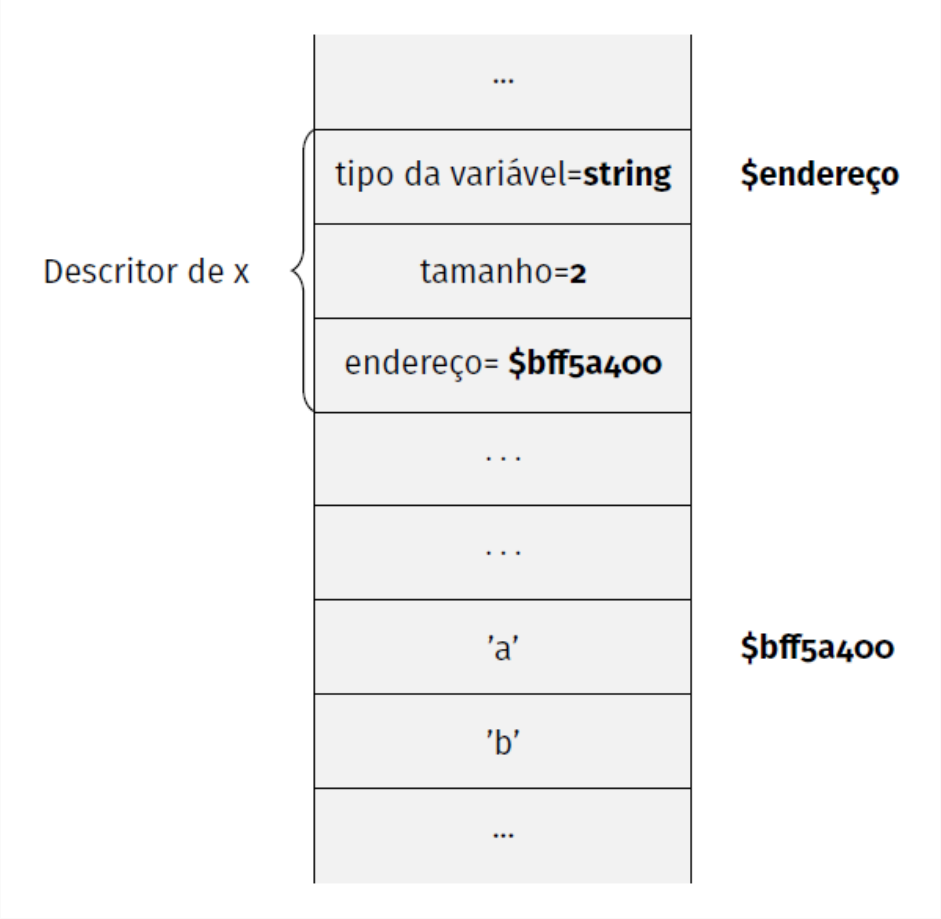
Porque é possível trocar o tipo de dado ?

- A linguagem Python é uma linguagem interpretada.
- Ou seja, um outro programa denominado interpretador coordena a execução das instruções.
- O interpretador mantem informações sobre todas as variáveis utilizadas pelo usuário.
- Cada símbolo associado a uma variável é na verdade um ponteiro (endereço) para uma área, denominada descritor, que contem informações sobre aquela variável.

- Por exemplo, o seguinte comando:

```
x='ab'
```

- cria inicialmente um descritor para x
- No descritor existe um endereço da memória aonde efetivamente está armazenado a string.



- Suponha agora que executa-se o seguinte comando:

```
x=2815
```

- Agora o interpretador atualiza o descritor de x indicando que agora a variável é um número inteiro além de indicar uma nova posição de memória aonde está o conteúdo da variável ($0AFF=2815).

Podemos descobrir o tipo da variável

- A existência de descritores permite que seja possível questionar o sistema sobre o tipo da variável através da função **type**( ) como indicado abaixo.

```
In [1]: x='ab'

In [2]: type(x)
Out[2]: str

In [3]: x=2815

In [4]: type(x)
Out[4]: int

In [5]:
```

Tipos de dados da linguagem Python

- Além das variávei simples do tipo número, string, boolean, etc. Python possui alguns tipos de variáveis que a caracterizam mais fortemente: listas, dicionários e tuplas.
- Já utilizamos listas na Aula de Lab 1 e no EP1.
- A seguir ilustramos o que seriam dicionários e tuplas.

Dicionários

- Dicionários contêm pares chaves-valores (*key-values*)
- Por exemplo o arquivo dicionario.py contêm o seguinte programa:

```
dict = {'Name': 'Maria', 'Age': 7, 'Class': 'First'}

print("dict['Name']: ", dict['Name'])
print("dict['Age']: ", dict['Age'])
print("dict['Class']: ", dict['Class'])
```

- O resultado da execução desse programa é ilustrado abaixo:

```
dict['Name']: Maria
dict['Age']: 7
dict['Class']: First
```

tuplas

- tuplas são sequencias de objetos imutáveis.
- A listagem a seguir (arquivo tupla.py) ilustra alguns exemplos de definições de tuplas:

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5 );
tup3 = "a", "b", "c", "d";
```

- repare no uso de parenteses e ponto e vírgula.
- parenteses podem ser omitidos.

Objetos na linguagem Python

Definição da classe Dog

- O arquivo dog.py contêm o seguinte código:

```
class Dog():
  """A simple attempt to model a dog."""

   def __init__(self, name, age):
      """Initialize name and age attributes."""
      self.name = name
      self.age = age

   def sit(self):
      """Simulate a dog sitting in response to a command."""
      print(self.name.title() + " is now sitting.")

   def roll_over(self):
      """Simulate rolling over in response to a command."""
      print(self.name.title() + " rolled over!")
```

Classe Dog

- A classe **Dog** possui três métodos.
- O primeiro método é denominado `__init__`

```
    def __init__(self, name, age):
        """Initialize name and age attributes."""
        self.name = name
        self.age = age
```

- Trata-se de um método especial denominado **construtor** na terminologia OOP.
- O método `__init__` é sempre executado quando o objeto é criado para realizar a inicialização das variáveis do objeto.
- As variáveis internas à classe sempre possuem o prefixo **self**.
- Dessa forma, as variáveis internas são: **self**`.name` e **self**`.age`.

Classe Dog

- Os outros dois métodos definem operações sobre o objeto.
- `sit()` indica que o cachorro está sentado através da impressão de uma mensagem.

```
    def sit(self):
        """Simulate a dog sitting in response to a command."""
        print(self.name.title() + " is now sitting.")
```

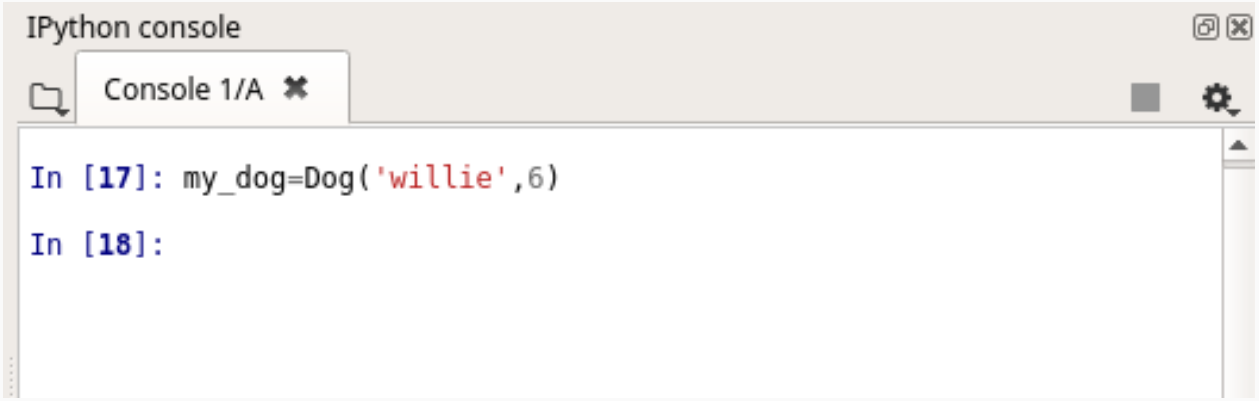
- `roll_over()` indica que o cachorro está rolando sobre o corpo através da impressão de uma mensagem.

```
    def roll_over(self):
        """Simulate rolling over in response to a command."""
        print(self.name.title() + " rolled over!")
```

- Observe que a função `title()` serve para que String seja formatada com letra inicial Maiúscula.

Classe Dog

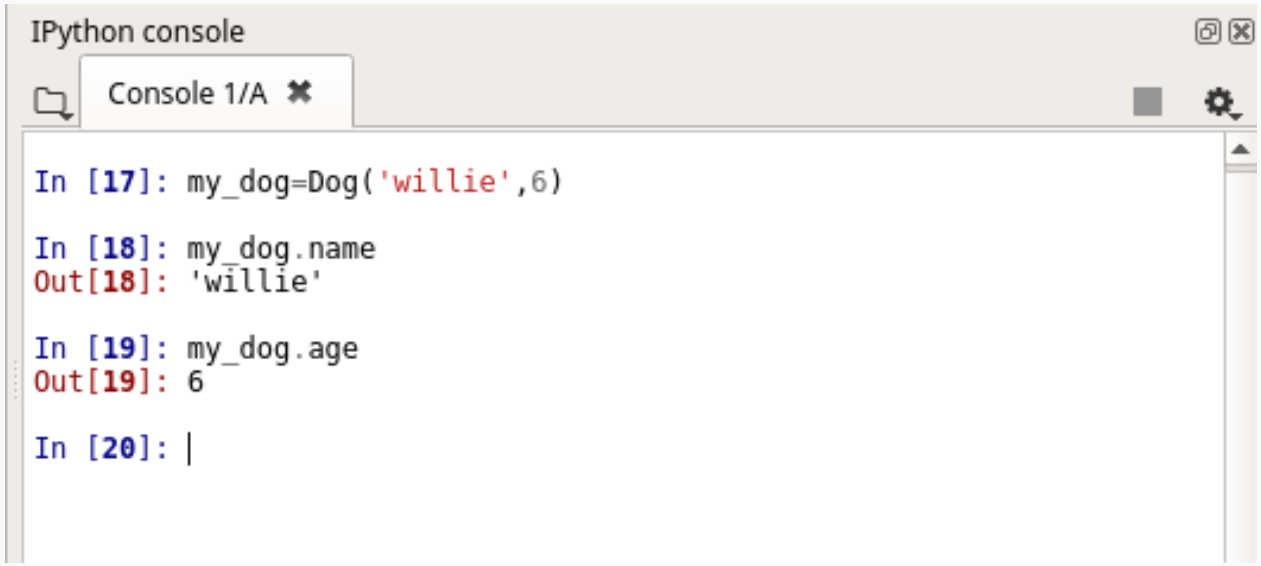
- Para utilizar a classe **Dog** carregue o arquivo dog.py na ide Spyder .
- Compile o arquivo.
- No console crie um objeto do tipo **Dog** como indicado abaixo:



- `my_dog` é um objeto do tipo **Dog**.
- Observe que ao executar `Dog('willie',6)` o sistema executa o método `__init__` correspondente à classe **Dog**.

Classe Dog

- É possível acessar as variáveis internas do objeto `my_dog` como indicado abaixo:



- Verifique tal comportamento digitando esses comandos no seu console.

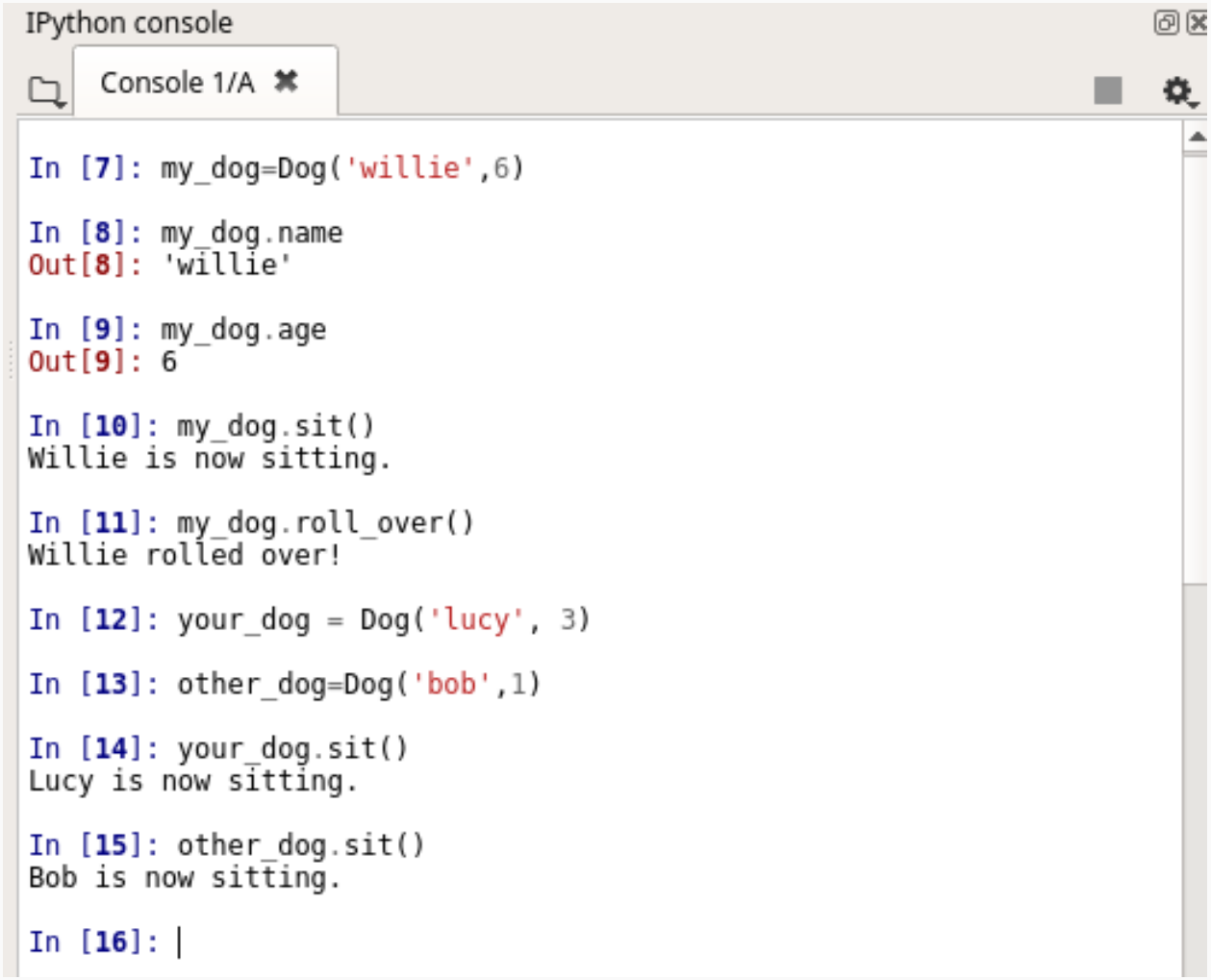
Classe Dog

- É possível executar os métodos correspondentes à classe **Dog** como indicado abaixo:

- Verifique tal comportamento digitando esses comandos no seu console.

Classe Dog

- É possível criar vários objetos do tipo **Dog**.
- Por exemplo, podemos criar o objetos `your_dog` e `other_dog` como indicado abaixo:

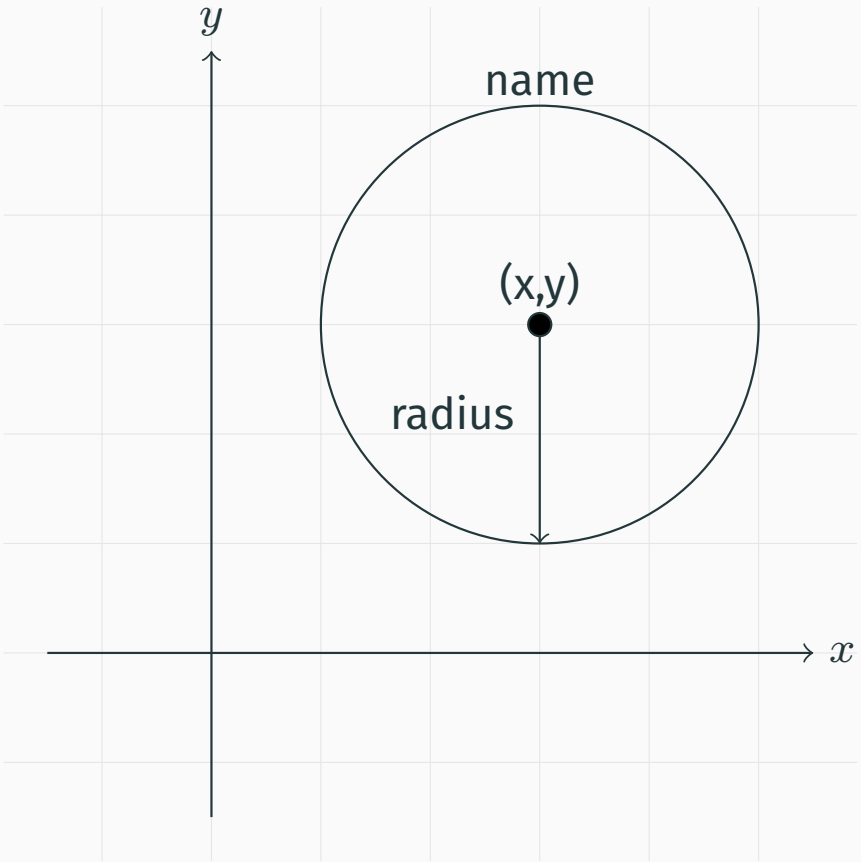


- Verifique tal comportamento digitando esses comandos no seu console.

Representação de círculos

Criando uma classe para representar círculos

- Deseja-se criar uma classe de objetos para representar figuras geométricas do tipo círculos.
- A classe círculos deve ser representada pelos seguintes campos: uma string que representa o nome do círculo, as coordenadas do centro do círculo e a medida do raio geométrico.
- Deseja-se também uma função que calcula a área do círculo.



Classe Circle

- Uma possível representação de uma classe **Circle** se encontra no arquivo circle0.py.
- Note que nessa definição os parãmetros de entrada possuem um valor **default**.
- Carregue o arquivo na IDE Spyder  e verifique o seu funcionamento.


```
import math
class Circle():
  def __init__( self, name = 'circle',x = 0, y = 0, radius = 0.0 ):
    self.name = name
    self.x = float( x )          # forcing to be float if it is not !
    self.y = float( y )
    self.radius = float( radius )

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

Classe Circle

- um pograma **main()** que demonstra a utilização da classe **Circle**.
- São criados dois objetos do tipo Circle: a e b.
- Note que o objeto b é criado com valores **default**.

```
def main():
    a=Circle('Circo',1,1,3)
    # examinando o conteudo interno do objeto a
    print('Conteudo interno do objeto a')
    print('a.name=',a.name)
    print('a.x=',a.x)
    print('a.y=',a.y)
    print('a.radius=',a.radius)
    print('Area do Circulo',a.name,'=',a.area(),'\n')

    # Cria objeto b, Circle com valores default
    b=Circle()
    # examinando o conteudo interno do objeto b
    print('Conteudo interno do objeto b - Valores default')
    print('b.name=',b.name)
    print('b.x=',b.x)
    print('b.y=',b.y)
    print('b.radius=',b.radius)
    print('Area do Circulo',b.name,'=',b.area(),'\n')
if __name__ == "__main__": main()
```

Objetos dentro de listas

- Os objetos podem ser colocados dentro de listas.
- O arquivo circle1.py contêm a listagem apresentada a seguir.
- Carregue o arquivo na IDE Spyder  e verifique o seu funcionamento.

```
import math
class Circle():
  def __init__( self, name = 'circle',x = 0, y = 0, radius = 0.0 ):
    self.name = name
    self.x = float( x )          # forcing to be float if it is not !
    self.y = float( y )
    self.radius = float( radius )

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

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
      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)
      print('cordenada x =',lista_de_circulos[k].x)
      print('cordenada y =',lista_de_circulos[k].y)
      print('raio =',lista_de_circulos[k].radius)
if __name__ == "__main__": main()
```

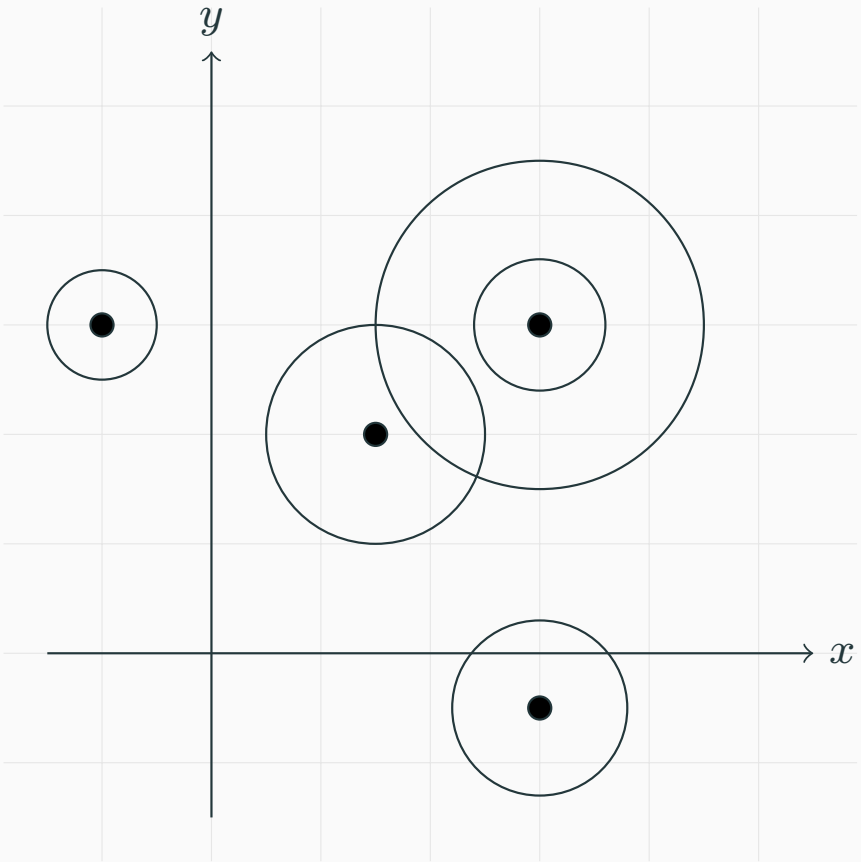
Objetos dentro de listas

- Podemos observar no programa **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)
  ```
- ```
  lista_de_circulos.append(a) # insere novo circulo na lista
  ```

Para você fazer

Verificar se os círculos possuem intersecção

- Deseja-se verificar se para um conjunto de círculos se existem círculos que possuem intersecção da área interna.



Verificar se os círculos possuem intersecção

- Carregue o arquivo circle2.py na IDE Spyder 


```
import math
class Circle():
  def __init__( self, name = 'circle',x = 0, y = 0, radius = 0.0 ):
    self.name = name
    self.x = float( x )          # forcing to be float if it is not !
    self.y = float( y )
    self.radius = float( radius )

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

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 = []
   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 nov\usepackage[nocolor]{drawstack}o circulo na lista

   # checa quais circulos possuem interseccoes com outros circulos
   # raiz quadrada - math.sqrt()

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

Verificar se os círculos possuem intersecção

- Projete um código na linguagem Python que verifique para cada círculo se existem intersecções com os outros círculos.
- Insira o código projetado no arquivo circle2.py.