



PMR3201 Computação para Automação

Aula de Laboratório 4

Programação Orientada a Objetos: Escopo, Pacotes, Tratamento de Exceções

Newton Maruyama
Thiago de Castro Martins
Marcos S. G. Tsuzuki
Rafael Traldi Moura
Andre Kubagawa Sato
21 de abril de 2020

PMR-EPUSP

1. Escopo de variáveis em objetos
2. Property
3. Tratamento de Exceções
4. Module × Package
5. Para você fazer

Escopo de variáveis em objetos

- ▶ O escopo de variáveis se refere ao local aonde as variáveis são válidas.
- ▶ O exemplo mais comum seria o conceito de variáveis definidas no programa principal `main()` e variáveis definidas localmente às funções.

- ▶ Por exemplo, no programa abaixo (arquivo teste.py) existem duas variáveis a.
- ▶ Ao final qual seria o valor de a da função main() ? Verifique executando o programa na IDE Spyder.

```
def altera(x):  
    a=5  
    y=5  
    y=y+1  
    x=x+1  
    return(x)  
  
def main():  
    a=9  
    x=0  
    y=2  
  
    altera(y)  
  
    print(a)  
    print(x)  
    print(y)  
if __name__ == "__main__": main()
```

Escopo de variáveis em objetos

- ▶ Como já apresentado em aula anterior o conceito de tipos abstratos de dados influenciou a evolução das linguagens de programação especialmente as linguagens orientadas a objetos como C++ e Java.
- ▶ São três os principais conceitos utilizados: abstração de dados (*data abstraction*), encapsulamento (*encapsulation*) e ocultamento de informações (*information hiding*).
- ▶ Dentro dessa perspectiva tanto atributos (variáveis internas) quanto métodos (funções) devem ter o seu escopo controlado.
- ▶ Por exemplo, as linguagens C++ e Java possuem modificadores de acesso `public`, `protected` e `private`.
- ▶ A utilização de modificadores permite bloquear o acesso direto às variáveis internas.
- ▶ O acesso somente é permitido se realizado através de métodos da interface com escopo controlado.

- ▶ Java utiliza modificadores de acesso para classes, variáveis, métodos e construtores. Os quatro níveis de acesso são enumerados a seguir:
 - ▶ Visível ao package (default) -> sem modificadores,
 - ▶ Visível à classe apenas -> `private`,
 - ▶ Visível ao pacote e todas às sub-classes -> `protected`,
 - ▶ Visível para o mundo -> `public`.

- ▶ Python não possui modificadores para controle de acesso.
- ▶ Entretanto, algumas convenções na construção do programa permitem obter o mesmo resultado.
- ▶ As seguintes convenções são usualmente utilizadas:
 - ▶ variáveis públicas: sem a utilização de símbolos especiais
Exemplo: x, y
 - ▶ variáveis protegidas: utilização de underscore _
Exemplo: _x, _y
 - ▶ variáveis privadas: utilização de duplo underscore __
Exemplo: __x, __y

Name mangling

- ▶ A utilização de double underscore é uma construção especial denominada Name Mangling.
- ▶ Verifique o próximo programa (arquivo CupPrivate.py) cujo código está ilustrado abaixo. Carregue e execute o programa na IDE Spyder.
- ▶ Note que a instrução <redCup.__content = 'tea'> não possui efeito.
- ▶ O acesso é feito através da instrução <redCup._Cup__content = 'tea'>.

```
# Private variable
class Cup:
    def __init__(self, color):
        self._color = color # protected variable
        self.__content = None # private variable
    def fill(self, beverage):
        self.__content = beverage
    def empty(self):
        self.__content = None
    def __str__(self):
        return('color = ' + str(self._color) + ' content = ' + str(self.__content))

redCup = Cup("red")
# Does this syntax work ?
redCup.__content = 'tea'
# Let's check !
print(redCup)

# If you want to have direct access to __content you need to use
# the following syntax
print('The correct way')
redCup._Cup__content = 'tea'
print(redCup)
```

- ▶ Independente da utilização do conceito de variáveis públicas, protegidas ou privadas usualmente muitos defendem a idéia de que a manipulação das variáveis internas só pode ser realizada através de uma interface composta por funções públicas.
- ▶ As funções da interface podem ler e alterar o valor das variáveis internas.
- ▶ Desta forma, objetos externos 'enxergam' outros objetos através de funções.
- ▶ Na terminologia da linguagem Python as funções que fazem a leitura de valores de variáveis internas são denominadas **getters** e as funções que alteram o valor das variáveis são denominadas **setters**

Classe Circle - getters e setters

- Uma implementação da classe **Circle** é apresentada a seguir (arquivo circleadt.py):

```
import math
class Circle():
    def __init__(self, name = 'circle', x = 0.0, y = 0.0, radius = 0.0):
        self.name = name
        self.x = float(x)
        self.y = float(y)
        self.radius = float(radius)

    # getters and setters
    def get_name(self):
        return(self.name)
    def set_name(self, name):
        self.name = name
    def get_x(self):
        return(self.x)
    def set_x(self, x):
        self.x = float(x)
    def get_y(self):
        return(self.y)
    def set_y(self, y):
        self.y = float(y)
    def get_radius(self):
        return(self.radius)
    def set_radius(self, radius):
        self.radius = float(radius)

    # modifier
    def area(self):
        return math.pi * self.radius ** 2
    def __str__(self):
        return('name = ' + self.name + ' x = ' + str(self.x) + ' y = ' +
            str(self.y) + ' radius = ' + str(self.radius))
```

- ▶ Ao final do arquivo circleadt.py existe uma função `main()` que ilustra a utilização da classe **Circle**.
- ▶ Carregue o arquivo na IDE Spyder e execute o programa.

```
def main():
    a = Circle() # Circle with default

    # lets set values
    a.set_name('Circulo o')
    a.set_x(10.0)
    a.set_y(5.0)
    a.set_radius(2.0)

    # lets get values
    print('name = ', a.get_name())
    print('x = ', a.get_x())
    print('y = ', a.get_y())
    print('radius = ', a.get_radius())

    # alternatively we can use __str__()
    print()
    print('alternativamente podemos fazer uso de __str__')
    print(a)

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

Property

Construindo interfaces com Property

- ▶ Na linguagem Python, **property()** é uma função que cria um objeto do tipo property.

```
property(fget=None, fset=None, fdel=None, doc=None)
```

- ▶ onde `fget()` é a função para obter o valor de um atributo,
- ▶ `fset()` é a função para 'setar' o valor de um atributo,
- ▶ `fdel()`, é a função para 'deletar' um atributo,
- ▶ `doc`, é uma string de comentário.

Exemplo: interface temperature

- ▶ Verifique abaixo a classe **Celsius** (Arquivo temperatureo.py).
- ▶ A variável interna que armazena a temperatura é `_temperature`
- ▶ Toda vez que existe a utilização de `temperature` aciona-se a interface, ou seja, utiliza-se `get_temperature` OU `set_temperature`.

```
class Celsius:
    def __init__(self, temperature = 0):
        self.temperature = temperature

    def to_fahrenheit(self):
        return (self.temperature * 1.8) + 32

    def get_temperature(self):
        print("Getting value")
        return self._temperature

    def set_temperature(self, value):
        if value < -273:
            raise ValueError("Temperature below -273 is not possible")
        print("Setting value")
        self._temperature = value

    temperature = property(get_temperature, set_temperature)
```

Teste da interface temperature

- A seguir apresentamos um programa que testa a interface temperature (arquivo testetemperatureo.py)

```
from temperatureo import Celsius
def main():
    a = Celsius()
    a.temperature = 50

    print('a = ',a.temperature)

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


- Abaixo temos a saída no console da IDE Spyder:



```
IPython console
Console 1/A ✖

In [6]: runfile('/home/maruyama/Documents/Graduacao/PMR3201/Slides/
Aula4/Codigo/testetemperature0.py', wdir='/home/maruyama/Documents/
Graduacao/PMR3201/Slides/Aula4/Codigo')
Reloaded modules: temperature0
Setting value
Setting value
Getting value
a = 50

In [7]:
```

- Note que a cada entrada no getter ou setter imprime-se a mensagem 'Setting value' ou 'Getting value'.

Uma outra maneira de declarar a interface

- Uma outra maneira equivalente de declarar a interface temperature é ilustrada abaixo utilizando @property e @temperature.setter (arquivo temperature1.py)

```
class Celsius:
    def __init__(self, temperature = 0):
        self._temperature = temperature

    def to_fahrenheit(self):
        return (self.temperature * 1.8) + 32

    @property
    def temperature(self):
        print("Getting value")
        return self._temperature

    @temperature.setter
    def temperature(self, value):
        if value < -273:
            raise ValueError("Temperature below -273 is not possible")
        print("Setting value")
        self._temperature = value
```

Colocando interfaces em Circle

- ▶ Podemos construir interfaces para os vários *getters* e *setters*.
- ▶ Exemplo no arquivo circleadtproperty.py

```
import math
class Circle():
    def __init__(self, name = 'circle', x = 0.0, y = 0.0, radius = 0.0):
        self.__name = name
        self.__x = float(x)
        self.__y = float(y)
        self.__radius = float(radius)

    # getters and setters
    def get_name(self):
        return(self.__name)
    def set_name(self, name):
        self.__name = name
    name = property(get_name, set_name)

    def get_x(self):
        return(self.__x)
    def set_x(self, x):
        self.__x = float(x)
    x = property(get_x, set_x)
```

```
def get_y(self):  
    return(self.__y)  
def set_y(self,y):  
    self.__y = float(y)  
y = property(get_y,set_y)  
  
def get_radius(self):  
    return(self.__radius)  
def set_radius(self,radius):  
    self.__radius = float(radius)  
radius = property(get_radius,set_radius)  
  
# modifier  
def area(self):  
    return math.pi * self.radius ** 2  
  
def __str__(self):  
    return( 'name = ' + self.name + ' x = ' + str(self.x) + ' y = ' +  
           str(self.y) + ' radius = ' + str(self.radius) )
```

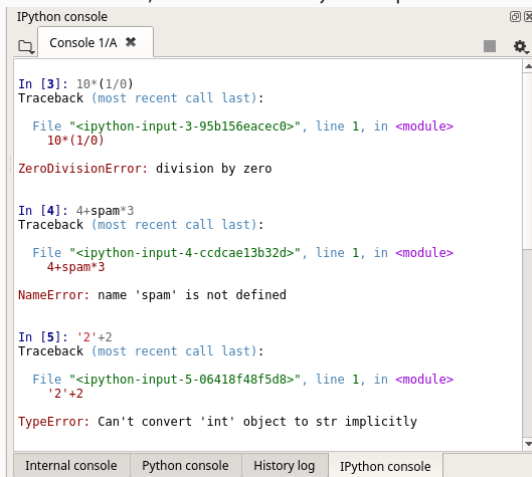
- O programa main ilustra a utilização das interfaces da classe **Circle**

```
def main():  
    a = Circle() # Circle with default values  
  
    # in the following we use the interface concept  
    # lets set values  
    a.name = 'Circulo o'  
    a.x = 10.0  
    a.y = 5.0  
    a.radius = 2.0  
  
    # lets get values  
    print('name = ',a.name)  
    print('x = ',a.x)  
    print('y = ',a.y)  
    print('radius = ',a.radius)  
  
    # alternatively we can use __str__()  
    print()  
    print('alternativamente podemos fazer uso de __str__')  
    print(a)  
  
if __name__ == "__main__": main()
```

Tratamento de Exceções

Tratamento de exceções

- ▶ Erros detectados durante a execução do programa são denominados exceções.
- ▶ Provavelmente, muitos de vocês já se depararam com as seguintes exceções:



```
IPython console
Console 1/A ✖

In [3]: 10*(1/0)
Traceback (most recent call last):
  File "<ipython-input-3-95b156eacec0>", line 1, in <module>
    10*(1/0)
ZeroDivisionError: division by zero

In [4]: 4+spam*3
Traceback (most recent call last):
  File "<ipython-input-4-ccdcae13b32d>", line 1, in <module>
    4+spam*3
NameError: name 'spam' is not defined

In [5]: '2'+2
Traceback (most recent call last):
  File "<ipython-input-5-06418f48f5d8>", line 1, in <module>
    '2'+2
TypeError: Can't convert 'int' object to str implicitly
```

Internal console Python console History log IPython console

Exceções definidas pelo usuário

- ▶ O usuário pode definir suas próprias classes de exceção tornando-as subclasses da classe pré-definida **Exception**
- ▶ A seguir apresentamos duas definições de classes de exceção:

```
# define Python user-defined exceptions
class Error(Exception):
    """Base class for other exceptions"""
    pass

class ValueTooSmallError(Error):
    """Raised when the input value is too small"""
    pass

class ValueTooLargeError(Error):
    """Raised when the input value is too large"""
    pass
```

- ▶ Utilizando essas classes constroe-se um programa (arquivo testeExcecaoUsuario.py) aonde o usuário tenta descobrir qual o número pré-definido (no caso number=10).
- ▶ Note que são utilizados comandos **raise** e **try except**.
- ▶ Quando o comando **raise** é executado a exceção é capturada pelo **except** correspondente.
- ▶ Utiliza-se a declaração **finally**, verifique o seu funcionamento.

- Carregue o arquivo testeExcecaoUsuario.py na IDE Spyder e verifique o seu funcionamento.

```
# define Python user-defined exceptions
class Error(Exception):
    """Base class for other exceptions"""
    pass

class ValueTooSmallError(Error):
    """Raised when the input value is too small"""
    pass

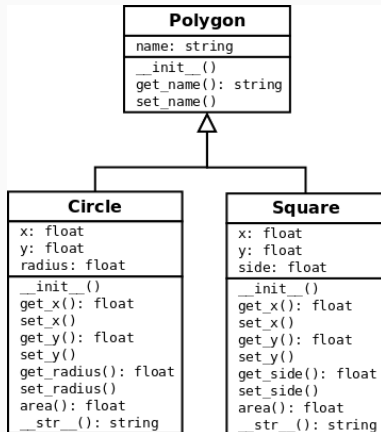
class ValueTooLargeError(Error):
    """Raised when the input value is too large"""
    pass

number = 10

while True:
    try:
        i_num = int(input("Enter a number: "))
        if i_num < number:
            raise ValueTooSmallError
        elif i_num > number:
            raise ValueTooLargeError
        break
    except ValueTooSmallError:
        print("This value is too small, try again!")
        print()
    except ValueTooLargeError:
        print("This value is too large, try again!")
        print()
    finally:
        print('I am always here !')
print("Congratulations! You guessed it correctly.")
```

Module × Package

- ▶ Módulos na linguagem Python se referem a arquivos contendo funções e classes que podem ser importados por outro programa.
- ▶ Por exemplo, o arquivo `polygonproperty.py`, contém a seguinte hierarquia de classes:
- ▶ A superclasse **Polygon** contém apenas a variável `name`.
- ▶ As Subclasses **Circle** e **Square** implementam getters and setters além de interfaces correspondentes.



- Um outro programa `main()`, no arquivo `testmodule.py`, faz uso das classes definidas no arquivo `polyproperty.py` como ilustrado abaixo:

```
from polygonproperty import Circle, Square

def main():
    a = Circle() # Circle with default values

    a.name = 'Circulo o'
    a.x = 5.0
    a.y = 3.0
    a.radius = 1.0

    print(a)

    b = Square()
    b.name = 'Square o'
    b.x = 2.0
    b.y = 7.0
    b.side = 2.0
    print(b)

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

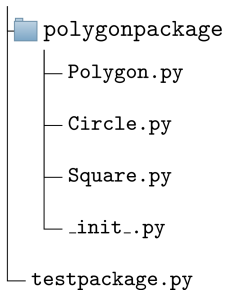
- Carregue na IDE Spyder os arquivos `polygonproperty.py` e `testmodule.py` e verifique seu funcionamento.

- ▶ Nesse exemplo a hierarquia de classes do exemplo anterior é transformada num package denominado **polygonpackage**.
- ▶ Para criar um package inicialmente cria-se um folder denominado **polygonpackage**.
- ▶ Dentro desse folder foram colocados os arquivos das classes: Polygon.py, Circle.py e Square.py além de um arquivo obrigatório denominado `__init__.py`. O conteúdo desse arquivo é ilustrado a seguir:

```
from .Polygon import *  
from .Square import *  
from .Circle import *
```

- No mesmo nível aonde está o folder **polygonpackage** um programa é definido que utiliza o package (arquivo testpackage.py).

Python



- Carregue o programa na IDE Spyder e verifique o seu funcionamento.

```
from polygonpackage import Circle
from polygonpackage import Square

def main():
    a = Circle() # Circle with default values

    a.name = 'Circulo o'
    a.x = 5.0
    a.y = 3.0
    a.radius = 1.0

    print(a)

    b = Square()
    b.name = 'Square o'
    b.x = 2.0
    b.y = 7.0
    b.side = 2.0
    print(b)

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

Para você fazer

Tratamento de exceções

- ▶ A seguir apresentamos um programa com tratamento de exceções (arquivo excecao_poligono.py)
- ▶ Uma exceção denominada NonIdentifiedPolygon foi criada quando o tipo de polígono não é identificado.

```
from polygonproperty import Circle, Square

class Error(Exception):
    pass
class NonIdentifiedPolygon(Error):
    pass

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', 'circulo', 1.0, 2.0, 3.0], ['square', 'quadrado', 2.0, 2.0, 1.0],
                                ['triangle', 2.0, 1.0, 5.0]]

    numero_de_poligonos = len(parametros_dos_poligonos)
```

```

for k in range(numero_de_poligonos):
    try:
        if parametros_dos_poligonos[k][0]=='circle':
            a = Circle()
            a.name = parametros_dos_poligonos[k][1]
            a.x = parametros_dos_poligonos[k][2]
            a.y = parametros_dos_poligonos[k][3]
            a.radius = parametros_dos_poligonos[k][4]
            lista_de_poligonos.append(a) # insere novo circulo na lista
        elif parametros_dos_poligonos[k][0]=='square':
            a = Square()
            a.name = parametros_dos_poligonos[k][1]
            a.x = parametros_dos_poligonos[k][2]
            a.y = parametros_dos_poligonos[k][3]
            a.side = parametros_dos_poligonos[k][4]
            lista_de_poligonos.append(a) # insere novo square na lista
        else:
            raise NonIdentifiedPolygon
    except NonIdentifiedPolygon:
        print()
        print(parametros_dos_poligonos[k][0])
        print('Poligono nao identificado !')
    else:
        print()
        print('Poligono processado corretamente !')
    finally :
        print('Sempre passo por aqui !')

```

```
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('\nIsto e um poligono do tipo Circle')
    else:
        print('\nIsto e um poligono do tipo Square')
        print(lista_de_poligonos[k])
        print('area =', lista_de_poligonos[k].area())

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

- ▶ Modifique o programa apresentado no arquivo `excecaopoligono.py` como solicitado.
- ▶ Criar uma classe de exceção denominada **InadequateParameters**.
- ▶ Essa exceção deve ser executada quando os parâmetros para a criação dos polígonos é inadequada, ou seja, tanto o número de parâmetros quanto o tipos dos parâmetros devem ser verificados.