UFCD 10794 - Programação avançada em Python

Notebook 10 - POO em Python

Sílvia Martins

AEFHP 2022

Classes e objetos em Python

Como podemos aplicar no Python, programação estruturada e programação orientada a objetos, podemos usar num mesmo programa, classes e funções.
Uma caraterística de POO no Python são as suas inúmeras bibliotecas que reutilizam classes predefinidas.Componentes que são úteis para implementação de determinadas funcionalidades e que promovem processos de reutilização de *software*.

- Uma **classe** permite a definição de um objeto (**instância da classe**) e a **reutilização** desse objeto várias vezes e em diferentes contextos num programa em Python.
- A **classe** é o elemento estrutural para o desenvolvimento de *software* orientado a**objetos**.
- As classes e as instâncias permitem representar objetos do mundo real de uma forma mais expressiva quando comparado com a aplicação de funções (paradigma procedimental).
- Os **objetos** têm **atributos (estrutura)**, que os caraterizam e **métodos** que definem os seu comportamento.
- As classes organizam-se em **packages** (pacotes) e **módulos**, sendo um package um conjunto de módulos e um módulo um conjunto de funções.

Visibilidade das variáveis

Em Python tanto as variáveis de instância como as variáveis estáticas são **públicas**, isto é, os seus valores podem ser lidos ou manipulados fora das classes que as **encapsulam**. Porém, para evitar erros de programação, o Python permite "esconder" as variáveis, fazendo com que elas não sejam diretamente visíveis para outras classes ou módulos.

As variáveis cujo nome é precedido por **duplo sublinhado (underscore)** não são visíveis para outras classes ou módulos de terceiros, sendo consideradas fortemente escondidas. Há ainda a possibilidade de esconder menos as variáveis, fazendo preceder o seu nome por um **único sublinhado (underscore)** . Contudo, estas variáveis "forte" ou "fracamente escondidas" podem ser sempre sê-lo, através da modificação do seu acesso.

Alguns detalhes a ter em conta na definição de uma classe em Python:

- As especificações de uma classe não requer parêntesis no final da designação.Os parêntesis são somente necessários na especificação de funções e de métodos, e na criação de uma instância;
- O primeiro argumento de um método é o termo `self` *. Utilizado para identificar e referir uma instância;
- Na definição dos atributos (Ex: *self.Nome*),guardados em variáveis de instância, a variável deve ser precedida com o termo `self` , ou seja, é utilizado para identificar e referir uma instância da classe;

Após a definição de uma classe, os seus membros (atributos e métodos) são utilizados e manipuladosnos programas através da sintaxe *class.member*

A atribuição de valores e respetiva alteração dos atributos pode ser feita:

- Através dos métodos dentro da classe;
- Diretamente nas instâncias da classe;
- no corpo de interação dos programas através da sintaxe: instancia.atributo = valor (como no exemplo que se segue)

*O termo `self` representa uma referência para o objeto que está a ser istanciado. Em Java, por exemplo, é usado o termo `_this_`

Exemplos de criação de classes e instâncias de uma classe:

São instanciados 2 objetos (x e y) de uma classe designada minhaClasse1:

In [7]:

```
#definição de uma classe sem estrutura
class minhaClasse1:
    pass     # significa que não terá mais especificações (membros, isto é, atributos ou mé

#instância da classe
P1=minhaClasse1() #associa-se o nome da instância ao nome da classe

print(P1)


#definir dinâmicamente atributos x e y
P1.x=40
P1.y=24
print("x, y= ",P1.x,",", P1.y)
```

```
<__main__.minhaClasse1 object at 0x000001E7FDBB4580>
x, y=  40 , 24
```

Métodos

As ações dos objetos e as interações entre eles definem o comportamento das instâncias de uma classe.
O método em Python é elemento fundamental para determinar o comportamento de uma classe. Em Python um método é uma função cuja assinatura tem o seguinte formato:

- Nome do método;
- Lista de parâmetros de entrada entre parêntesis curvos.

Os métodos são públicos, mas como as varíáveis, podem ser fraca ou fortemente "escondidos" de terceiros, iniciando o seu nome por duplo sublinhado ou por um só sublinhado. No entanto, classes terceiras podem sempre invocar os métodos ainda que estejam "escondidos".

Métodos de instância e métodos estáticos

Os métodos de instância definem funcionalidades dos objetos, portanto, só podem ser invocados/chamados se se associarem aos repetivos objetos.
Como foi dito anteriormente, o primeiro parâmetro de um método de uma instância é sempre *self* , sendo obrigatório.

Sintaxe:

```
#definição (instruções, código) do método
...
```

As classes também podem conter `métodos estáticos` , isto é, métodos que fazem parte da classe, mas que não se referem aos objetos.
Por exemplo, uma classe pode conter métodos para atribuir valores ou imprimir os valores das variáveis estáticas.

Exemplos:

`Método de instância` **ClassFinal** que calcula a classificação final de um aluno de Informática que realizou 2 testes:

In [ ]:

```
def ClassFinal(self):
    return (self.Teste1 + self.Teste2) / 2
```

`Método estático` **TotalAlunos** , da classe **ClassFinal** para atribuir a uma disciplina o nº total de alunos inscritos:

In [ ]:

```
def TotalAlunos(N):
    EstudanteInf.TotalAlunos=N
```

A classe EstudanteInf contém 1 método com os argumentos Nome, Teste1 e Teste2, que atribui valores ao objeto *self* , definindo assim 3 `variáveis de instância` - **Nome**, **Teste1** e **Teste2**

In [5]:

```
class EstudanteInf:
    # atributos
    def set_EstudanteInf(self, Nome, Teste1, Teste2):
        self.Nome=Nome
        self.Teste1=Teste1
        self.Teste2=Teste2

    def TotalAlunos(N):
        EstudanteInf.TotalAlunos=N
```

Os métodos de instância têm de ser invocados para um objeto da classe que os define, enquanto que os métodos estáticos são invocados para a classe que os contém.

Exemplo:

Demonstração da invocação dos métodos de instância **set_EstudanteInf** e estático **Totalalunos** da classe **EstudanteInf**.

In [6]:

```
#programa principal(interface)
#criar 1 instância
E1=EstudanteInf()

#invocação do método de instância
E1.set_EstudanteInf('Ana',12,14)  # objeto.nome_metodo
print(E1.Nome)

#invocação do método estático
EstudanteInf.TotalAlunos(20)    #classe.nome_metodo
```

```
Ana
```

Método construtor *init*

Um construtor é um método específico para criar e inicializar uma instância de uma classe. O construtor de uma classe Python é definido através do método *init* . Ele irá inicializar o valor dos atributos da classe precedidos da instrução *self*
A assinatura do cosntrutor em Python é a seguinte:
`def __init__ (self, p1, p2, ..., pn):` sendo *p1, p2, ... pn* a lista de parâmetros que recebe os valores a atribuir aos objetos da classe respetiva.

Exemplo:

A classe EstudanteInf define 3 `variáveis de instância` - **Nome**, **Teste1** e **Teste2** que agora estão fortemente escondidas. É usado um método construtor :

In [7]:

```
class EstudanteInf:
    N='a'
# método construtor
    def __init__(self, N, T1, T2):
        self.Nome = N
        self.Teste1 = T1
        self.Teste2 = T2
# método de instância
    def reset(self):
        self.Nome=''
        self.Teste1 = 0
        self.Teste2 = 0
# método de instância
    def ClassFinal(self):
        return (self.Teste1 + self.Teste2) / 2
```

In [8]:

```
E=EstudanteInf('Ana', 14, 16 )
print(f"{E.Nome} ")
print(f"{E.Nome} teve a classif. final de {E.ClassFinal()} valores")
E.reset()
print(E.Teste1)
```

```
Ana
Ana teve a classif. final de 15.0 valores
0
```

Os **atributos da classe ou variáveis e classe** diferem dos atributos de dados (variáveis de instância) na medida em que podem ser acedidos sem criar uma instância dessa classe:

In [ ]:

```
print(EstudanteInf.N)
```

Conjunto de atributos de classe predefinidos (build in class attributes)

Estes atributos determinam informação da própria classe, representando os atributos embutidos numa classe Python . Sintaxe: .

- `__dict__` Dicionário que contém o conjunto de nomes (namespace) da classe.
- `__doc__` String de documentação (texto/comentários) da classe /*None* se não estiver definida).
- `__name__` Nome da classe
- `__module__` Nome do módulo no qual a classe foi definida, em modo (ambiente Shell) interativo, o atributo é designado por `__main__`
- `__bases__` Tuplo qeu contem as classes-base (superclasses). apresenta um tuplo vazio se não existirem classe-base

Podemos redefinir a classe sem atributos desde que identifiquemos a classe com o `__main__`

Exemplo:

In [9]:

```
class EstudanteInf:
#método construtor
    def __init__(self, N, T1, T2):
        self.Nome = N
        self.Teste1 = T1
        self.Teste2 = T2
        #métodos
    def reset(self):
        self.Nome=''
        self.Teste1 = 0
        self.Teste2 = 0

    def ClassFinal(self):
        return (self.Teste1 + self.Teste2) / 2

#programa principal
if __name__=="__main__":
    E=EstudanteInf('Ana', 14, 16 )
    print(f"{E.Nome} ")
    print(f"{E.Nome} teve a classif. final de {E.ClassFinal()} valores")
    E.reset()
    print(E.Teste1)
```

```
Ana
Ana teve a classif. final de 15.0 valores
0
```

Mais utilizações de atributos de classe:

In [10]:

```
class contador:
    # atributos (membros) da classe
    cont=0
    soma=0
    #método construtor
    def __init__(self):
        self.__class__.cont += 1 #aplica atributo __class__

#    Programa principal
#variáveis pre-definidas da classe
print(contador.__doc__)
print(contador.__name__)
print(contador.__module__)

# atributos da classe
c=contador()
print(c.cont)
c.soma=25
print(c.soma)
```

```
None
contador
__main__
1
25
```

Encapsulamento

O mecananismo de `encapsulamento` permite esconder do utilizador os detalhes de implementação de uma classe. Em ambientes de programação, o objetivo do encapsulamento e da criação de **membros privados** é indicar ao programador que não tem acesso direto aos atributos da classe, devendo ser feito através dos métodos disponíveis.
É uma forma eficiente de proteger os dados manipulados dentro da classe, além de determinar onde esta classe poderá ser manipulada. Todos os atributos deverão ser privados - é desejável que os atributos de um objeto só possam ser alterados poe ele mesmo, inviabilizando situações imprevistas Habitualmente os métodos de um objeto são public.

Para definir os **membros privados** (já dito no início deste nb) aplica,os um prefixo, dois underscores (__) às variáveis (atributos) e aos métodos. Desta forma os atributos e métodos não estarão visíveis fora do âmbito da definição da classe.

Exemplo:

In [11]:

```
class contador:
    # atributos (membros) da classe
    __cont=0    # variável privada
    soma=20    # variável pública

    def contador_publico(self):
        print("...método público")

    def __contador_privado(self):
        print("...método privado")

    def icount (self):
        self.__cont+= 1
        print(self.__cont)
ic=contador()
print(ic.soma)
print(ic.icount())
ic.contador_publico()
ic._contador__contador_privado() # _ para baceder ao método privado é necessário preceder o
```

```
20
1
None
...método público
...método privado
```

Acesso e propriedades

As variáveis de instância e as variáveis estáticas podem ser manipuladas, por qualquer método, de qualquer classe, através dos seus acessos públicos. Estes sõa métodos para ler e alterar os valores das variáveis de instância e das variáveis estáticas cuja visibilidade foi escondida.
Regra geral, as classes definem, para cada uma das suas variaveis de instância, um par de acessos publicos:

- acesso para leitura - *getter* - que devolve o valor corrente da variável de instância e que pode também incluir código de formatação e saída de dados;
- Acesso para atribuir um novo valor - *setter* - à variável de instância e que pode também incluir um código de validação de dados.

Exemplo:

In [2]:

```
class EstudanteInf:
# método construtor
    def __init__(self, N, T1, T2):
        self.__Nome = N      # os atributos são privados e podem ser acedidos atravéas das
        self.__Teste1 = T1
        self.__Teste2 = T2
    #getter
    def GetNome(self):     # a palavra get que precede a variável serve apenas para melhor
        return self.__Nome
    #setter
    def SetNome(self, N):     # a palavra set que precede a variável serve apenas para melh
        self.__Nome=N
```

Este par de acessos pode ser substituído por uma propriedade com as duas funcionalidades de *setter* e *getter*, recorrendo-se a uma propriedade para ler e alterar o valor da varíavel de instância Nome de EstudanteInf.

Exemplo:

In [3]:

```
class EstudanteInf:
# método construtor
    def __init__(self, N, T1, T2):
        self.__Nome = N      # os atributos são privados e podem ser acedidos através das p
        self.__Teste1 = T1
        self.__Teste2 = T2
    @property
    def Nome(self):
        return self.__Nome
    @Nome.setter
    def Nome(self, N):
        self.__Nome=N
```

Assim, aquando da invocação externa , o par de acessos será tratado como se fossem um só atributo visível do objeto:

In [4]:

```
if __name__=="__main__":
    E=EstudanteInf("", 0, 0)
    E.Nome=input("Insere um nome->")
    print(E.Nome)
```

```
Insere um nome->Ana
Ana
```

In [5]:

```
class EstudanteInf():
    def __init__(self, N, T1, T2):
        self.__Nome=N
        self.__Nota1=T1
        self.__Nota2=T2

    @property
    def Nome(self):
        return self.__Nome
    @Nome.setter
    def Nome(self, N):
        self.__Nome=N
    @property
    def Nota1(self):
        return self.__Nota1
    @Nota1.setter
    def Nota1(self, T1):
        self.__Nota1=T1
    @property
    def Nota2(self):
        return self.__Nota2
    @Nota2.setter
    def Nota2(self, T2):
        self.__nota2=T2

    def classFinal(self):
        return (self.__Nota1+self.__Nota2)/2
    def Inpressao(self):
        print(f"{'Nome':^15}{'Teste1':^10}{'Teste2':^10}{'classif. Final':^15}")
        print(f"{'self.Nome':^15}{'self.Nota1':^10}{'self.Nota2':^10}{'self.ClassFinal()':^
        return
```

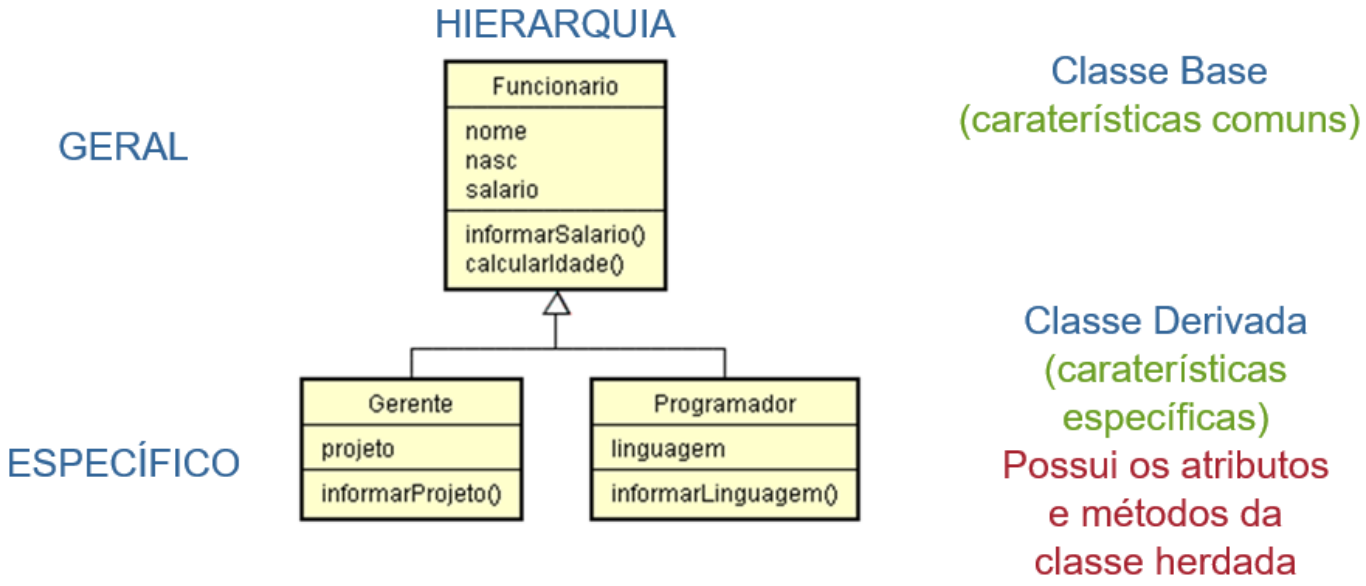
Herança

As classes são estruturadas hierarquicamente em `superclasses` , `classes` e `subclasses` por recurso à `herança` . As classes `herdam` os atributos/propriedades e os métodos das superclasses de que derivam e acrescentam mais atributos e métodos e passam tudo às subclasses que delas derivam. Esta hierárquia é representada graficamente por árvores (figura seguinte). Uma vez que uma subclasse é uma `especialização` da superclasse de que deriva, as árvores de classes mostram os diferentes níveis de `generalização` .

Sintaxe:

```
class nome_da_ classe (Nome_da_superclasse) :
    def __init__ (self, p1, p2, …Pm, Pm+1…Pn):
        Nome_da_superclasse.__init__(self,P1, …Pm):
            self.Pn=pm+1
             …
            self.Pn=Pn
```

Nota: O nome da superclasse pode ser substituído pelo método *`super()`* .



Exemplos de criação de uma subclasse B que herda os atributos e métodos da classe A:

In [ ]:

```
#supondo a classe A
class A:
    def __init__ (self, n, s):
        self.Nome=n  #atributos da superclasse(A) Nome e Salario
        self.salario=s

class B(A):
    def __init__(self, n, s, p): # construtor da subclasse(B)
        A.__init__(self, n, s)  #atributos herdados da superclasse(A)
        self.ProjetoB=p         #atributo específico da subclasse(B)
```

Herança múltipla

O Python permite a herança múltipla, isto é, uma subclasse pode derivar de mais do que uma superclasse.

Exemplo de criação de da classe Candidato que herda os atributos e métodos das classe UPorto e ULondon:

In [7]:

```
#superclasse UPorto
class UPorto:
    def __init__(self, fac, disc):
        self.Fac=fac  #atributos da superclasse(A) Nome e Salario
        self.Disciplina=disc

#superclasse ULondon
class ULondon:
    def __init__(self, colegio, curso, area):
        self.Colegio=colegio  #atributos da superclasse(A) Nome e Salario
        self.Curso=curso
        self.Area=area
#subclasse Candidato
class Candidato(UPorto, ULondon):
    def __init__(self, fac, disc, colegio, curso, area, n): # construtor da subclasse(Candi
        UPorto.__init__(self, fac, disc)  #atributos herdados das superclasses(UPorto, ULon
        ULondon.__init__(self, colegio, curso, area)
        self.Nome=n
```

Polimorfismo

`Polimorfismo` significa, em programação, muitas formas e traduz-se por:

- Um `método` com o `mesmo nome` e a `mesma` lista de parâmetros (`assinatura`), poder ser implementado em muitas classes com funcionalidades diferentes;
- Um método com o mesmo nome e a mesma assinatura poder ser implementado numa hierarquia de classes, de modo que o método mais próximo do objeto se sobrepõe aos outros e, portanto, é executado aquando da sua invocação;
- Objetos diferentes que implementam um método com as mesma assinatura poderem ser tratados como iguais pelas funções que invocam esses métodos.

Por vezes também se usa a palavra `sobrecarga` (em inglês, `overload` ou `override`) para referir esta caraterística. Pode-se dizer que a execução das versões do método irá depender da instância da classe que for criada.

Exemplo:

In [8]:

```
#superclasse
class FiguraGeometrica:
    def __init__(self, nome):
        self.nome=nome

    def calcula_area(self):   # método virtual
        pass
#classe Quadrado
class Quadrado(FiguraGeometrica):
    def __init__(self, lado):
        FiguraGeometrica.__init__(self,"QUADRADO")
        self.lado=lado

    def calcula_area(self):
        return self.lado **2

#classe triângulo
class Triangulo(FiguraGeometrica):
    def __init__(self, base, altura):
        FiguraGeometrica.__init__(self,"TRIÂNGULO")
        self.base = base
        self.altura = altura

    def calcula_area(self):
        return (self.base * self.altura)/2
#programa principal

quadrado = Quadrado(2)
triangulo = Triangulo (2, 5)

print(quadrado.calcula_area())
print(triangulo.calcula_area())
```

```
4
5.0
```

Bibliografia

Caleiro.C., Ramos.J.(2016). Notebook 10 - Programação orientada a objectos Dep. Matemática.nLisboa:IST
Vasconcelos, J. (2015).Python - Algoritmia e Programação Web. Lisboa:FCA
Carvalho. Adelaide. (2021). Práticas de Python - Algoritmia e programação. Lisboa: FCA
https://www.youtube.com/watch?v=ymYDQylb78Y (https://www.youtube.com/watch?v=ymYDQylb78Y)
https://docente.ifrn.edu.br/brunogurgel/disciplinas/2012/fprog/aulas/poo/POO4-Heranca-I.pdf
(https://docente.ifrn.edu.br/brunogurgel/disciplinas/2012/fprog/aulas/poo/POO4-Heranca-I.pdf)