Fundamentos de Programação @ LEIC/LETI

Semana 11

Programação com objetos

Classes e objetos. Encapsulação. Exemplo complexos. Anonimato. Polimorfismo. Herança. Composição. Exemplos

Alberto Abad, Tagus Park, IST, 2018

Introdução

Classes em Python

- Em Python podemos definir classes da seguinte forma (em BNF):

```
<definição de classe> ::= class <nome> {(<nome>)}: NEWLINE
    INDENT <definição de método> + DEDENT

<definição de método> ::= def <nome> (self{, <parâmetros formais>)}: NEWLINE
    INDENT <corpo> DEDENT
```

- `self` é uma referência ao próprio objeto utilizada para referenciar os atributos e métodos do objeto.

Objetos em Python

- Um objecto é uma **instância** de uma classe.
- Podemos instanciar objetos da seguinte forma (BNF) e assinar a nomes de variaveis:

```
<instanciação de objeto> ::= <nome_da_classe>({<parâmetros>})
<assinação objeto> ::= <nome_do_objeto> "=" <instanciação de objeto>
```

- Após de criado um objeto, podemos invocar os seus métodos e atributos como se fosse um nome composto. Em BNF:

```
<acesso_atributos> ::= <nome_do_objeto>.<nome_do_atributo>
<acesso_métodos> ::= <nome_do_objeto>.<nome_do_método>({<parâmetros>})
```

Classes e objetos em Python - Exemplo Pessoa

- Para definir uma classe simples, basta com definir um único método: o **construtor** `__init__`:

In [197]:
```
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade
    def fala(self):
        return self.nome + ": Bla bla bla"
```

- Agora, podemos criar/instanciar um objeto do tipo Pessoa e interagir come ele (métodos e atributos):

In [200]:
```
p1 = Pessoa("Alberto", 25)
p1.nome
```

Out[200]:
```
'Alberto'
```

- Python reconhece o tipo como se fosse *built-in* (`type`, `isinstance`, etc.)

Exemplo Complexos

In [201]:

```
class complexo:
    def __init__(self, x, y):
        if not(isinstance(x, (int, float)) and isinstance(y, (int, float))):
            raise ValueError('complexo: argumentos invalidos, x e y tem de ser num
eros')
        self.real = x
        self.imaginario = y

    def parte_real(self):
        return self.real

    def parte_imaginaria(self):
        return self.imaginario

    def e_zero(self):
        return self.zero(self.real) and self.zero(self.imaginario)

    def e_imaginario_puro(self):
        return self.zero(self.real) and not self.zero(self.imaginario)

    def igual(self, other):
        if not isinstance(other, complexo):
            raise ValueError("")
        return self.zero(self.real - other.real) and \
                self.zero(self.imaginario - other.imaginario)


    def para_string(self):
        return str(self.real) + "+" + str(self.imaginario) + "i"

    def zero(self, x):
        return abs(x) < 0.0000001
```

Exemplo Complexos

Alteração 1 - Tipo imutável → Tipo mutável

- O nosso tipo complexo é imutável (só fornece seletores), mas podemos definir um tipo mutável (com modificadores):

In [217]:

```
class complexo:
    def __init__(self, x, y):
        if not(isinstance(x, (int, float)) and isinstance(y, (int, float))):
            raise ValueError('complexo: argumentos invalidos, x e y tem de ser num
eros')
        self.real = x
        self.imaginario = y

    ## Novos modificadores
    def actualizar_parte_real(self, x):
        self.real = x

    def actualizar_parte_imaginaria(self, y):
        self.imaginario = y

    ## Resto da classe
    def parte_real(self):
        return self.real

    def parte_imaginaria(self):
        return self.imaginario

    def e_zero(self):
        return self.zero(self.parte_real()) and self.zero(self.parte_imaginaria())

    def e_imaginario_puro(self):
        return self.zero(self.parte_real()) and not self.zero(self.parte_imaginari
a())

    def igual(self, w):
        if not isinstance(w, complexo):
            raise ValueError('complexo: igual: z tem de ser complexo')
        return self.zero(self.parte_real() - w.parte_real()) \
           and self.zero(self.parte_imaginaria() - w.parte_imaginaria())

    def para_string(self):
        return str(self.parte_real()) + '+' + str(self.parte_imaginaria()) + 'i'
```

Exemplo Complexos

Alteração 1 - Tipo imutável → Tipo mutável

In [36]:
```
z = complexo(10,20)
w = complexo(10, 20)
print(z.igual(w))
z.actualizar_parte_real(5)
print(z.igual(w))
print(w.para_string())
print(z.para_string())
```

```
True
True
10+20i
10+20i
```

Exemplo Complexos

Problema - Anonimato da representação!?!?

- Que acontece se fazemos `w.real`? ou `w.imaginario`!?!?

In [223]:
```
w = complexo(5, 20)
print(w.para_string())
w.real = (10, 34)
print(w.para_string())
```

```
5+20i
(10, 34)+20i
```

- Em linguagens OO (como C++ ou Java) fazemos *data hiding* restringindo o acesso aos atributos e métodos de uma classe:
  - `public`: acessível por qualquer classe
  - `protected`: só acessível pela própria classe ou sub-classes
  - `private`: só acessível pela própria classe

Anonimato: Restrição de acesso por convenção

- Em Python não existe *data hiding* e na prática a restrição de acesso é *realizada* por **convenção**:

```
class teste_classe:
  def __init__(self):
      self.public = "public"
      self._protected = "protected"
      self.__private = "private"
```

In [228]:
```
class teste_classe:
    def __init__(self):
        self.public = "public"
        self._protected = "protected"
        self.__private = "private" #name mangling

t = teste_classe()
t._teste_classe__private
```

Out[228]:
```
'private'
```

Exemplo Complexos

Alteração 2 - Restrição de acesso: Métodos/atributos privados

In [233]:

```
class complexo:
    def __init__(self, x, y):
        if not(isinstance(x, (int, float)) and isinstance(y, (int, float))):
            raise ValueError('complexo: argumentos invalidos, x e y tem de ser num
eros')
        self.__real = x  ## alterar
        self.__imaginario = y ## alterar

    def parte_real(self):
        return self.__real ## alterar

    def parte_imaginaria(self):
        return self.__imaginario ## alterar

    def actualizar_parte_real(self, x):
        if not isinstance(x, (int, float)):
            raise ValueError('complexo: argumentos invalido, x tem de ser numero')
        self.__real = x ## alterar

    def actualizar_parte_imaginaria(self, y):
        if not isinstance(x, (int, float)):
            raise ValueError('complexo: argumentos invalido, x tem de ser numero')
        self.__imaginario = y ## alterar

    def e_zero(self): ## alterar
        return self.__zero(self.parte_real()) and self.__zero(self.parte_imaginari
a())

    def e_imaginario_puro(self): ## alterar
        return self.__zero(self.parte_real()) and not self.__zero(self.parte_imagi
naria())

    def igual(self, w): ## alterar
        if not isinstance(w, complexo):
            raise ValueError('complexo: igual: z tem de ser complexo')
        return self.__zero(self.parte_real() - w.parte_real()) \
           and self.__zero(self.parte_imaginaria() - w.parte_imaginaria())
```

Polimorfismo

- Anteriormente, já verificámos que muitas operações em Python são sobrecarregadas i.e., aplicam-se a vários tipos de dados:
  - Ex. a operação + que podemos aplicar a inteiros, reais, strings, tuplos, listas, etc.
- As operações que podem ser aplicadas sobre diferentes tipos de dados dizem-se **polimórficas**.
- Em Python, podemos utilizar polimorfismo para estender muitas das operações pré-definidas como `__add__` (+), `__sub__` (–), `__mul__` (*), `__truediv__` (/), `__eq__` (==).
- Também podemos sobrecarregar métodos como `__init__`, `__repr__` e `__str__` (transformação para string, i.e, representação externa).

Polimorfismo: Sobrecarrega de métodos

Python permite personalizar objetos definindo alguns métodos com nomes especiais:

- **`__init__`**: Construtores sobrecarregados com diferente número de parâmetros. Permite as diferentes variantes de passo de parâmetros: posicional, *default*, *keyword*, número variável.
- **`__repr__`**: retorna uma string adequada para o programador. É chamado pelo interpretador em modo interativo.
- **`__str_`**: retorna uma string adequada para o utilizador. É chamado pela função *built-in* `str()`

In [76]:
```
    def __init__(self, x=0, y=0):
        if not(isinstance(x, (int, float)) and isinstance(y, (int, float))):
            raise ValueError('complexo: argumentos invalidos, x e y tem de ser num
eros')
        self.real = x
        self.imaginario = y

    def __repr__(self):
        return self.para_string()
```

Polimorfismo: Sobrecarrega de operadores

Operator	Expression	Internally
Addition	p1 + p2	`p1.__add__(p2)`
Subtraction	p1 - p2	`p1.__sub__(p2)`
Multiplication	p1 * p2	`p1.__mul__(p2)`
Power	p1 ** p2	`p1.__pow__(p2)`
Division	p1 / p2	`p1.__truediv__(p2)`
Floor Division	p1 // p2	`p1.__floordiv__(p2)`
Remainder (modulo)	p1 % p2	`p1.__mod__(p2)`
Bitwise Left Shift	p1 << p2	`p1.__lshift__(p2)`
Bitwise Right Shift	p1 >> p2	`p1.__rshift__(p2)`
Bitwise AND	p1 & p2	`p1.__and__(p2)`
Bitwise OR	p1 | p2	`p1.__or__(p2)`
Bitwise XOR	p1 ^ p2	`p1.__xor__(p2)`
Bitwise NOT	~p1	`p1.__invert__()`

Polimorfismo: Sobrecarrega de operadores

- Vejamos como utilizar polimorfismo para estender a operação + para o tipo complexo, implementando o método `__add__`:

```
def __add__(self, z):
        if not isinstance(z, complexo):
            raise ValueError('complexo: argumentos invalido, z tem de ser complex
o')
        return complexo(self.parte_real() + z.parte_real(),\
                        self.parte_imaginaria() + z.parte_imaginaria())
```

Polimorfismo: Sobrecarrega de operadores de comparação

Operator	Expression	Internally
Less than	p1 < p2	`p1.__lt__(p2)`
Less than or equal to	p1 <= p2	`p1.__le__(p2)`
Equal to	p1 == p2	`p1.__eq__(p2)`
Not equal to	p1 != p2	`p1.__ne__(p2)`
Greater than	p1 > p2	`p1.__gt__(p2)`
Greater than or equal to	p1 >= p2	`p1.__ge__(p2)`

Mais informação em: https://www.programiz.com/python-programming/operator-overloading (https://www.programiz.com/python-programming/operator-overloading)

Polimorfismo: Sobrecarrega de operadores de comparação

- Vejamos como utilizar polimorfismo para estender a operação de comparação == para o tipo complexo, implementando o método `__eq__`:

```
def __eq__(self, z):
        if not isinstance(z, complexo):
            raise ValueError('complexo: argumentos invalidos, z tem de ser comple
xo)
        return self.igual(z)
```

Exemplo Complexos

Alteração 3 - Métodos e operadores polimórficos

In [237]:

```
class complexo:
    def __init__(self, x=0, y=0):
        if not(isinstance(x, (int, float)) and isinstance(y, (int, float))):
            raise ValueError('complexo: argumentos invalidos, x e y tem de ser num
eros')
        self.__real = x
        self.__imaginario = y

    def parte_real(self):
        return self.__real

    def parte_imaginaria(self):
        return self.__imaginario

    def actualizar_parte_real(self, x):
        if not isinstance(x, (int, float)):
            raise ValueError('complexo: argumentos invalido, x tem de ser numero')
        self.__real = x

    def actualizar_parte_imaginaria(self, y):
        if not isinstance(x, (int, float)):
            raise ValueError('complexo: argumentos invalido, x tem de ser numero')
        self.__imaginario = y

    def e_zero(self):
        return self.__zero(self.parte_real()) and self.__zero(self.parte_imaginari
a())

    def e_imaginario_puro(self):
        return self.__zero(self.parte_real()) and not self.__zero(self.parte_imagi
naria())

    def igual(self, w):
        if not isinstance(w, complexo):
            raise ValueError('complexo: igual: z tem de ser complexo')
        return self.__zero(self.parte_real() - w.parte_real()) \
           and self.__zero(self.parte_imaginaria() - w.parte_imaginaria())
```

```
    def para_string(self):
        return str(self.parte_real()) + '+' + str(self.parte_imaginaria()) + 'i'

    def actualizar_parte_real(self, x):
        if isinstance(x, (int,float)):
            raise ValueError('complexo: argumento invalido, x tem de ser numero')
        self.__real = x

    def actualizar_parte_imaginaria(self, y):
        if isinstance(y, (int,float)):
            raise ValueError('complexo: argumento invalido, y tem de ser numero')
        self.__imaginario = y

    def __zero(self, x):
        return abs(x) < 0.0000001

    def __repr__(self):
        return self.para_string()

    def __add__(self, z):
        if not isinstance(z, complexo):
            raise ValueError('complexo: z tem de ser complexo')
        x = self.__real + z.__real # Podemos fazer isto para z?
        y = self.__imaginario + z.parte_imaginaria()
        return complexo(x, y)

    def __eq__(self, z):
        return self.igual(z)
```

Na próxima aula...

- ### Herança
- ### (Composição?)
- ### Mais exemplos

Herança

- A **PO** facilita a reutilização de componentes em diferentes programas.
- Em particular, a **herança** permite-nos especializar classes, através da definição de subclasses:
  - Uma classe (geralmente chamada de superclasse) é herdada por outra classe (geralmente chamada de subclasse).
  - A subclasse adiciona alguns atributos e métodos à superclasse.
- Em Python (BNF):

```
<definição de subclasse> ::= class <sublclasse nome> (<superclasse nome>): NEWLIN
E
    INDENT <definição de método> + DEDENT
```

Herança - Exemplo Pessoa, Professor, Aluno

In [446]:
```
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade
    def falar(self):
        return "Bla bla bla"
    def __repr__(self):
        return self.nome + '(' + str(self.idade) + ')'
```

In [485]:

```
class Professor(Pessoa):
    ## 1- Construtor
    def __init__(self, nome, idade, disciplina):
        super().__init__(nome, idade)
        self.disciplina = disciplina


    ## 2- Acrescentemos um metodo ensinar
    def ensinar(self):
        return "Ensina muitas coisas"

    def falar(self):
        return "O professor esta a falar: " + super().falar()
    # 5a - Especializemos o metodo falar
    # 5b - Especializemos o metodo falar baseado no metodo da super-classe

class Aluno(Pessoa):
    # 3- Construtor, o aluno tem um atributo adicional que é o numero de aluno
    def __init__(self, nome, idade, numero):
        super().__init__(nome, idade)
        self.numero = numero

    # 4- Acrescentemos um metodo aprender
    def aprender(self):
        return "Aprende algumas coisas"


    # 5a - Especializemos o metodo falar
    # 5b - Especializemos o metodo falar baseado no metodo da super-classe
```

Herança em Python

- O `isinstance()` retorna *True* se o elemento testado é do mesmo tipo ou de um tipo que é sublasse

- Para verificar se um tipo é subclasse:

```
issubclass(Professor, Pessoa)
```

- Para chamar ao construtor da superclasse:

```
super().__init__({parametros})
```

- Em geral, para referenciar métodos/atributos da superclasse utilizamos:

```
super().metodo({parametros})
```

In [495]:
```
issubclass(Pessoa, Professor)
```

Out[495]:
```
False
```

Herança e composição

- Em PO, se diz que a relação entre duas classes é de herança se é uma relação **IS-A**:
  - O Professor **é** uma Pessoa; O Aluno **é** uma Pessoa
- Nem todas as relações na modelização de um problema são do tipo IS-A
- Em PO existem outros mecanismos para a reutilização de código (relações entre classes).
- Em particular, a compossição é um outro tipo de relação que responde a **HAS-A**:
  - Um Professor pode **ter** Alunos

Herança e composição - Exemplo Pessoa, Professor, Aluno e Docente

In [496]:

```
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade
    def falar(self):
        return "Bla bla bla"

class Professor(Pessoa):
    def __init__(self, nome, idade):
        super().__init__(nome, idade)

    def ensinar(self):
        return "Estou a ensinar muitas coisas"

    def falar(self):
        return "Professor fala: " + super().falar()

    def __repr__(self):
        return "Professor " + self.nome

class Aluno(Pessoa):
    def __init__(self, nome, idade, numero):
        super().__init__(nome, idade)
        self.numero = numero

    def aprender(self):
        return "Estou a aprender algumas coisas"

    def falar(self):
        return "Aluno fala: " + super().falar()

    def __repr__(self):
        return "Aluno " + self.nome + '(' + str(self.numero) + ')'
```

Herança e composição - Exemplo Pessoa, Professor, Aluno e Docente

- **Objetivo**: Definir a entidade Docente que é um Professor é que têm Alunos

In [502]:

```
class Docente(Professor):
    ## 1 - Construtor - vamos fazer a lista de alunos privada
    def __init__(self, nome, idade):
        super().__init__(nome, idade)
        self.__alunos = []

    # 2 - Metodo para acrescentar alunos que aceite elementos individuais ou lista
s
    def acrescenta_alunos(self, alunos):
        if isinstance(alunos, Aluno):
            self.__alunos.append(alunos)
        elif isinstance(alunos, list) and all(isinstance(x,Aluno) for x in alunos
):
            self.__alunos.extend(alunos)

    # 3 - Metodo para mostras os alunos (str)
    def mostra_alunos(self):
        return str(self.__alunos)

    # 4 - Metodo para inserir nota de um aluno, acrescentaremos um novo atributo _
_notas

    # 5 -metodo para obter a nota de um aluno
```

Programação com objetos

Herança e composição - Exemplo Agência de viagens

- Consideremos uma agência de viagens onde são oferecidos vários produtos.
- Cada produto é caracterizado por um preço e uma designação.
- A um produto podemos também associar uma descrição.
- Dois produtos são iguais se tiverem o mesmo nome.

In [359]:

```
class av_produto:
    def __init__(self, nome, preco):
        if not(isinstance(nome, str) and isinstance(preco, (int, float))):
            raise ValueError('av_produto: argumentos invalidos')
        self.__nome = nome
        self.__preco = preco
        self.__descricao = None

    def get_nome(self):
        return self.__nome

    def set_nome(self, nome):
        if not isinstance(nome, str):
            raise ValueError('av_produto: set_nome: argumento invalido')
        self.__nome = nome

    def get_preco(self):
        return self.__preco

    def set_preco(self, preco):
        if not isinstance(preco, (int, float)):
            raise ValueError('av_produto: set_preco: argumento invalido')
        self.__preco = preco

    def get_descricao(self):
        return self.__descricao

    def set_descricao(self, descricao):
        if not isinstance(descricao, str):
            raise ValueError('av_produto: set_descricao: argumento invalido')
        self.__descricao = descricao

    def __repr__(self):
        return self.get_nome() + ' [preco: ' + str(self.get_preco()) + ']'

    def __eq__(self, produto):
```

```
        if not isinstance(produto, av_produto):
            raise ValueError('av_produto: __eq__: argumento invalido')
        return self.get_nome() == produto.get_nome()
```

Programação com objetos

Herança e composição - Exemplo Agência de viagens

- Nem todos os produtos nesta agência são iguais, temos:
  - viagens (que têm um local de origem e destino),
  - hotéis (que ficam num local e têm categorias),
  - circuitos (que podem incluir vários dos outros produtos).
- Para representarmos estes diferentes produtos devemos utilizar abstracções e reutilizar código.
- Podemos utilizar herança, subclasses, compossição e redefinição de métodos quando necessário.

Herança e composição - Exemplo Agência de viagens

In [360]:
```
class av_viagem(av_produto):
    def __init__(self, nome, preco, origem, destino):
        super().__init__(nome, preco)
        if not(isinstance(origem, str) and isinstance(destino, str)):
            raise ValueError('av_viagem: argumentos invalidos')
        self.__origem = origem
        self.__destino = destino

    def get_origem(self):
        return self.__origem

    def set_origem(self, origem):
        if not isinstance(origem, str):
            raise ValueError('av_viagem: set_origem: argumentos invalidos')
        self.__origem = origem

    def get_destino(self):
        return self.__destino

    def set_destino(self, destino):
        if not isinstance(destino, str):
            raise ValueError('av_viagem: set_destino: argumentos invalidos')
        self.__destino = destino

    def __repr__(self):
        return super().__repr__() + ' ' + self.get_origem() + ' -> ' + self.get_de
stino()
```

In [434]:
```
v1 = av_viagem("Viagem 1", 500, "Lisboa", "Porto")
v2 = av_viagem("Viagem 2", 2000, "Lisboa", "Rio de Janeiro")
v1 == v2
```

Out[434]:
```
False
```

Herança e composição - Exemplo Agência de viagens

In [371]:
```
class av_hotel(av_produto):
    def __init__(self, nome, preco, local, categoria):
        super().__init__(nome, preco)
        if not(isinstance(local, str) and isinstance(categoria, int)):
            raise ValueError('av_hotel: argumentos invalidos')
        self.__local = local
        self.__categoria = categoria

    def get_local(self):
        return self.__local

    def set_local(self, local):
        if not isinstance(local, str):
            raise ValueError('av_hotel: set_local: argumentos invalidos')
        self.__local = local

    def get_categoria(self):
        return self.__categoria

    def set_categoria(self, categoria):
        if not isinstance(categoria, int):
            raise ValueError('av_hotel: set_categoria: argumentos invalidos')
        self.__categoria = categoria

    def __repr__(self):
        return super().__repr__() + ' ' + self.get_local() + ' (' + str(self.get_c
ategoria()) + ' estrelas)'
```

In [436]:
```
h1 = av_hotel("Hotel P1", 200, "Porto", 4)
h2 = av_hotel("Hotel P2", 100, "Porto", 2)
h2
```

Out[436]:
```
Hotel P2 [preco: 100] Porto (2 estrelas)
```

Herança e composição - Exemplo Agência de viagens

In [376]:
```
class av_circuito(av_produto):
    def __init__(self, nome):
        super().__init__(nome, 0)
        self.__produtos = []

    def get_produtos(self):
        return tuple(self.__produtos)

    def add_produto(self, produto):
        if not isinstance(produto, av_produto):
            raise ValueError('av_circuito: add_produto: argumento invalido')
        self.__produtos.append(produto)

    def set_preco(self, preco):
        pass

    def get_preco(self):
        total = 0
        for x in self.get_produtos():
            total += x.get_preco()
        return total

    def __repr__(self):
        res = super().__repr__() + ' circuito:'
        for x in self.get_produtos():
            res += '\n\t' + str(x)
        return res
```

In [437]:
```
c1 = av_circuito("Circuito 1")
c1.add_produto(v1)
c1.add_produto(v2)
c1.add_produto(h1)
c1
```

Out[437]:
```
Circuito 1 [preco: 2700] circuito:
        Viagem 1 [preco: 500] Lisboa -> Porto
        Viagem 2 [preco: 2000] Lisboa -> Rio de Janeiro
        Hotel P1 [preco: 200] Porto (4 estrelas)
```