

Programación II

Tema 1

Paradigmas da programación
Programación orientada a obxectos

Paradigmas de programación

- Paradigma designa a unha forma de considerar algo. Paradigma de programación é un enfoque para a programación
- Tradicionalmente considéranse dous grupos de paradigmas:
 - 1) Imperativo: o programa contén instrucións que indican ao ordenador cómo cambiar o seu estado (variables). Inclúe dous paradigmas:
 - a) Procedimental: agrupa as instrucións en procedimentos.
 - b) Orientado a obxectos: agrupa as instrucións coa parte do estado do ordenador
 - 2) Declarativo: decláranse as propiedades do resultado, pero non cómo calculalo
 - a) Lóxico: o resultado é a resposta a unha pregunta sobre unha base de feitos e regras
 - b) Funcional: o resultado obtense como unha serie de aplicacións dunha función

Linguaxes de programación por paradigmas

- 1) Procedimentais: C, Fortran, Pascal, Cobol, Python
 - 2) Orientados a obxectos: Java, Smalltalk, Eiffel. C++ e Python permiten programar neste paradigma, pero non o forzan
 - 3) Lóxicos: Prolog, Lisp. Curry permite programación lóxica
 - 4) Funcionais: Lisp, Scheme, Haskell, ML y SQL. Python e Julia teñen características funcionais
- Varias linguaxes de programación proporcionan elementos de distintos paradigmas

Paradigma procedimental

- O programa é unha secuencia de procedimentos contendo instrucións: asignación, cálculo, entrada, saída, selección e iteración
- Cada procedemento actúa sobre as súas variables locais e os seus argumentos (datos), retornando valores (resultados)
- Poden existir datos globais, accesibles dende tódolos procedimentos
- Existe un procedemento principal, no que comeza e remata a execución, que chama aos demais
- Encapsulación limitada: pode haber acceso involuntario e indebido a datos dende fóra do seu ámbito
- Dependencia dos tipos de datos: dificultade para crear tipos abstractos

Paradigma orientado a obxectos

- O programa é unha colección de obxectos que interaccionan entre eles mediante **paso de mensaxes** que modifican o seu estado
- Definición de **clases**, grupos de obxectos do mesmo tipo. Permite clasificar comportamentos distintos pero relacionados
- **Encapsulación** de datos e operacións dentro dun obxecto
- **Abstracción**: plantiñas de procedimentos (*templates*) que codifican unha operación sobre datos abstractos e son aplicables a distintos tipos de datos
- **Herdanza** de obxectos: permite especializar comportamentos e reutilizar código, evitando repeticións
- **Polimorfismo**: acceso a comportamentos distintos usando unha única interface

Paradigma lóxico (1/2)

- O programa é unha colección de declaracións ou predicados lóxicos (feitos e regras de inferencia) que define o resultado da función, non **cómo** conseguilo
- Estas regras combinan proposicións lóxicas cos operadores AND, OR, implicación, equivalencia e negación
- A execución do programa consiste en propagar os datos de entrada (feitos) pola colección de regras retornando unha serie de solucións posibles
- Resolución: inferencia simple a partir de dúas declaracións
- Unificación: comparación de patrones que decide a instanciación (solución) que se pode facer cos feitos as regras do programa
- Aplícase recurrentemente para obter tódalas posibles solucións
- Razoamento automatizado: sistemas expertos, procesamento da linguaxe natural, demostración de teoremas

Paradigma lóxico (2/2)

- Baseado na lóxica de primeiro orde (ou de predicados)
- Un predicado é unha regra na cal se se cumpre unha combinación de condicións ou feitos lóxicos (que poden certos ou falsos) permite inferir unha conclusión:

condición1 AND NOT condición2 OR condición3 $\rightarrow$ conclusión

- A combinación pode incluír os operadores lóxicos AND, OR, XOR e NOT, o condicional “se *condición* entón *conclusión*”, “se e so se *condición* entón *conclusión*”, “*ni feito1 nin feito2*”
- Tamén permite o cuantificador universal “*para todo x condición*” e existencial “*existe x tal que condición*”
- Algunhas linguaxes lóxicas incorporan elementos da lóxica borrosa (*fuzzy*)
- As linguaxes funcionais son linguaxes lóxicas baseadas no *cálculo lambda*, única teoría lóxica de orde superior que se demostrou que é computable

Paradigma funcional

- O programa é unha colección de funcións matemáticas, coas súas entradas e saídas
- As funcións son obxectos: todo o que se pode facer noutras linguaxes con datos, pódese facer nas linguaxes funcionais con funcións
- Combínanse usando condicións, recursividade e composición funcional como estruturas de control, desaniman o uso de asignacións de variables
- Enfocado en procesamento de listas, usando recursividade sobre sublistas no canto de bucles
- Evalúan expresións no canto de usar sentenzas nos linguaxes procedimentais
- Enfócase en **qué** se calcula, non en **cómo**
- Usan funcións de orde superior, que operan sobre funcións no canto de sobre datos

Outros paradigmas

- 1) **Simbólico:** o programa manexa fórmulas matemáticas como datos
- 2) **Diferenciable:** o programa pode ser diferenciado automaticamente, de modo simbólico ou numérico
 - a) Calquera programa que implemente unha función vectorial pódese descompoñer nunha lista de asignacións elementais
 - b) Cada asignación pode derivarse usando unha táboa de búsquedas
 - c) Estas derivadas combínanse usando a regra da cadea para crear gradientes e matrices xacobianas
- 3) **Literario:** variante da programación imperativa. O programa é un texto en prosa (ensaio) en formato hipertexto con documentación. Cercano á persoa que programa.

Programación orientada a objetos

Características (1/2)

- O código organízase en clases. Cada **clase** agrupa variables (**atributos**) e funcións (**métodos**)
- Os métodos da clase operan sobre os seus atributos, ademais de recibir argumentos
- **Herdanza**, ou reusabilidade de atributos e métodos dun obxecto (clase base) por parte doutro obxecto (clase derivada).
- **Encapsulación** dos datos (atributos de obxectos), que poden ser inaccesibles dende fóra da clase
- **Abstracción**: definir grupos que inclúen datos distintos pero certos atributos e métodos comúns

Características (2/2)

- **Polimorfismo:** permite acceder a distintos métodos dunha única forma
 - Relacionado coa herdanza: distintas clases derivadas redefinen, usando o mesmo nome, un método da clase base
 - Executando este método para as clases derivadas (unha única forma), o método executado é distinto para cada clase (distintos métodos)
- Podes crear variables do tipo da clase. Cada unha é un obxecto desa clase
- **Obxecto:** é unha instancia dunha clase.
- Exemplo: o tipo de datos enteiro é como se fose unha clase, creando unha variable enteira creas (instancias) un obxecto desa clase

Características (3/3)

- Os atributos (propiedades) definen o estado do obxecto
- Os métodos cambian o estado do obxecto
-

Vantaxes e inconvenientes

- Aumenta a reusabilidade do código, ao dificultar interaccións indesexadas entre funcións
- Pode introducir restriccións que limitan a flexibilidade na programación
-

Clase

- Agrupación de atributos e métodos
- Definición de clase en Python
- Os atributos poden inicializarse
- `__init__` é o inicializador ou constructor da clase: execútase cando se crea un obxecto desa clase. Non retorna nada
- Pode recibir outros argumentos **args** que inicializan os atributos (estado)
- **self** debe ser o primeiro argumento de tódolos métodos (poden recibir outros argumentos **args**)

```
class nome_clase
    atributo1 [=val1]
    atributo2 [=val2]
    def __init__(self,[args]):
        ...
    def __del__(self):
        ...
    def __str__(self):
        return s
    def metodo1(self,[args]):
        return val
    def metodo2(self,[args]):
        return val
```

Obxecto

- Instancia dunha clase
- Creación dun obxecto en Python: `x=nome_clase(args)`
- Chámase a unha función que ten o mesmo nome que a clase, pero execútase o constructor de clase `__init__`
- Pásanse os argumentos **args** esperados polo constructor
- Acceso aos atributos do obxecto: `x.atributo1`
- Execución dos métodos do obxecto: `x.metodo1(args)`. Non se lle pasa argumento para **self**
- Dentro do obxecto, **self**, designa ao propio obxecto. Un método pode usar outro método da clase con **self.metodo1(args)**
- `__str__` é un método que retorna unha cadea de caracteres representando ao obxecto. So ten argumento **self**

Obxecto

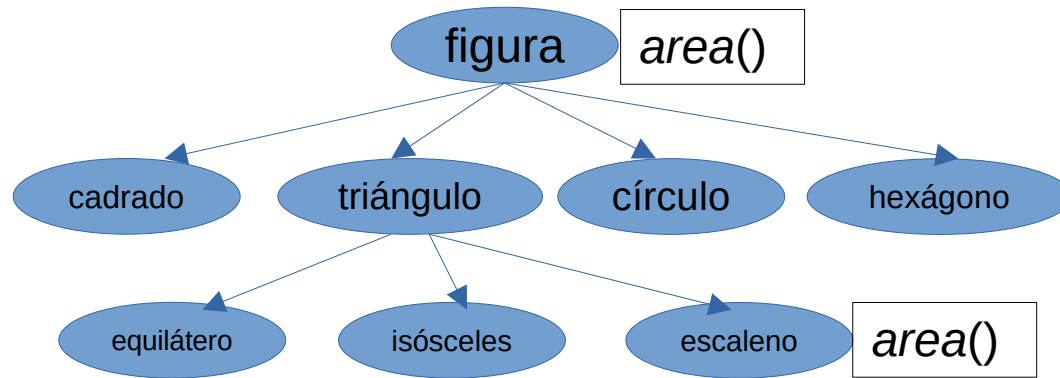
- O constructor debe reservar a memoria necesaria para os atributos do obxecto, inicializar estes atributos, abrir os ficheiros necesarios, etc
- O destructor **`__del__(self)`** debe encargarse de liberar a memoria, pechar os arquivos, entre outros
- Visibilidade dos atributos e métodos (encapsulación):
 - *public*: accesible dende calquera sitio (dentro e fóra da clase): comportamento por defecto
 - *private*: accesible so dende dentro da clase, non dende subclasses: variable que comeza con dobre subraiado. Ex: `__num`
 - *protected*: accesible dende a clase e subclasses: nome comeza por subraiado simple, ex. `_num`

Herdanza

- Consiste en definir unha clase derivada de outra (clase base)
- A clase derivada herda tódolos atributos e métodos da clase base, e pode engadir atributos e métodos
- Tamén pode redefinir (sobrecargar, *overload*) métodos da clase base. Redefinir significa usar o mesmo nome do método, pero cun código distinto que o método da clase base
- Cando se executan métodos redefinidos, execútase o método redefinido da clase derivada
- Cando se executa un método que a clase derivada non redefine, execútase o método da clase base

Xerarquía de clases

- Poden existir varios niveis de clases bases e derivadas
- Unha clase base pode ter varias clases derivadas, cada unha engadindo distintos atributos ou redefinindo métodos de formas distintas
- Exemplo de xerarquía de clases:



Herdanza en Python

- Defínese a clase base: **class** base
- Defínese a clase derivada como:
class derivada(base):
- Engádense atributos e métodos, e redefínense métodos a vontade
- Ás veces deíxanse baleiros os métodos da clase base que se redefinen nas clases derivadas, que son os únicos que se executarán. Pódese:
 - a) Non facer nada, pero non da erro: **pass**
 - b) Activar excepción: **raise NotImplementedError**
- A clase pasa a ser **abstracta**: define unha *interfaz*, ou serie de métodos das súas clases derivadas

```
class figura:  
    def __init__(self, xx):  
        self.x=xx  
    def __str__(self):  
        return 'Clase base figura'  
    def perimetro(self):  
        raise NotImplementedError  
    def area(self):  
        raise NotImplementedError  
    def volumen(self):  
        pass
```

```
class circulo(figura):  
    def __str__(self):  
        return 'Círculo de radio %g' % self.x  
    def perimetro(self):  
        return 2*pi*self.x  
    def area(self):  
        return pi*self.x**2  
    def volumen(self):  
        return 4/3*pi*self.x**3
```

Polimorfismo

- Poli+morfos=moitas formas
- Consiste en executar distintos métodos cun único nome
- ¿Cómo se fai?
 - 1) Defínense varias clases derivadas dunha clase base, que redefinen un método.
 - 2) Cando se executa o método dun obxecto da clase derivada, execútase o método que éste redefine
 - 3) Tódolos métodos redefinidos teñen o mesmo nome, polo tanto ...
 - 4) Executando os métodos redefinidos de distintas clases derivadas executamos métodos **distintos** usando o **mesmo** nome

Polimorfismo: exemplo en Python

```
from math import *
class figura:
    def __init__(self, xx):
        self.x=xx
    def __str__(self):
        return 'Clase base figura'
    def perimetro(self):
        raise NotImplementedError # lanza unha excepción
    def area(self):
        raise NotImplementedError
    def volumen(self):
        pass

#-----
class circulo(figura):
    def __str__(self):
        return 'Círculo de radio %g' % self.x
    def perimetro(self):
        return 2*pi*self.x
    def area(self):
        return pi*self.x**2
    def volumen(self):
        return 4/3*pi*self.x**3

#-----
class cadrado(figura):
    def __str__(self):
        return 'Cadrado de lado %g' % self.x
    def perimetro(self):
        return 4*self.x
    def area(self):
        return self.x**2
    def volumen(self):
        return self.x**3
```

```
class rectangulo(figura):
    def __init__(self, largo, ancho):
        figura.__init__(self, largo) #chamada o constructor de figura
        self.a=ancho # agregamos o novo atributo
    def __str__(self):
        return 'Rectangulo de lados %g e %g' % (self.x, self.a)
    def perimetro(self):
        return 2*self.x+2*self.a
    def area(self):
        return self.x*self.a
    def volumen(self):
        return self.a*self.a*self.x

#-----
figura=[circulo(1.2),cadrado(2.1),rectangulo(5, 2),figura(2)]
for fig in figura:
    try:
        print('fig=',fig)
        print('Perimetro: ',fig.perimetro(),' cm')
        print('Area: ',fig.area(),'cm^2')
        print('Volumen: ',fig.volumen(),' cm^3')
    except NotImplementedError:
        print('Clase base figura')
    print('-----')
```

```
In [1]: run polimorfismo.py
fig= Círculo de radio 1.2
Perimetro: 7.5398223686155035 cm
Area: 4.523893421169302 cm^2
Volumen: 7.238229473870882 cm^3
-----
fig= Cadrado de lado 2.1
Perimetro: 8.4 cm
Area: 4.41 cm^2
Volumen: 9.261000000000001 cm^3
-----
fig= Rectangulo de lados 5 e 2
Perimetro: 14 cm
Area: 10 cm^2
Volumen: 20 cm^3
-----
fig= Clase base figura
Clase base figura
```

Paradigmas de programación

Paso de mensaxes

- A execución de métodos dun obxecto por parte doutros obxectos na xerarquía de clases dun programa considérase como “**paso de mensaxes**” entre obxectos
- O paso dunha mensaxe cambia o estado do obxecto que a recibe
- Se unha clase derivada non ten redefinido o método, ascéndese pola xerarquía ata atopar unha clase que ten este método, que se executa
- O mensaxe é unha petición a un obxecto para realizar unha acción. A resposta do obxecto á petición depende dos métodos dispoñibles
- Obxectos de distintas clases responden de modo distinto á mesma mensaxe. Un obxecto novo pode substituír a outro existente sempre que responda ás mesmas mensaxes, facilitando o mantemento do código
- O cambio no paradigma procedimental esixe rescribir o procedemento, e manter a coherencia das súas entradas e saídas dificulta o mantemento

Sobrecarga de operadores (1/3)

- Redefinir operadores (matemáticos, asignación, etc.) a medida para datos especiales (clases)
- Exemplo: redefinir o operador + para vectores bidimensionais
- ¿Cómo se fai? Para sobrecargar un certo operador hai que definir un método cun nome especial
- Métodos especiais para sobrecarga de operadores **aritméticos binarios**:

Operador	+	-	*	/	//	%	**
Método	<code>__add__</code> (self,other)	<code>__sub__</code> (self,other)	<code>__mul__</code> (self,other)	<code>__truediv__</code> (self,other)	<code>__floordiv__</code> (self, other)	<code>__mod__</code> (self,other)	<code>__pow__</code> (self,other)

Sobrecarga de operadores (2/3)

- Métodos para sobrecarga de operadores de **comparación**:

Operador	<	>	<=	>=	==	!=
Método	<code>__lt__</code> (self,other)	<code>__gt__</code> (self,other)	<code>__le__</code> (self,other)	<code>__ge__</code> (self,other)	<code>__eq__</code> (self, other)	<code>__ne__</code> (self,other)

- Métodos para sobrecarga de operadores de **asignación**:

Operador	-=	+=	*=	/=	//=	%=	**=
Método	<code>__isub__</code> (self,other)	<code>__iadd__</code> (self,other)	<code>__imul__</code> (self,other)	<code>__itruediv__</code> (self,other)	<code>__iflloordiv__</code> (self, other)	<code>__imod__</code> (self,other)	<code>__ipow__</code> (self,other)

- Métodos para sobrecarga de operadores **unarios**:

Operador	=	+	~
Método	<code>__neg__</code> (self)	<code>__pos__</code> (self)	<code>__invert__</code> (self)

Sobrecarga de operadores (3/3)

Exemplo en Python:

```
class Punto2D(object):
    def __init__(self, x, y):
        self.x=x; self.y=y
    def __add__(self, v):
        return Punto2D(self.x+v.x, self.y+v.y)
    def __sub__(self, v):
        return Punto2D(self.x-v.x, self.y-v.y)
    def __mul__(self, s):
        return Punto2D(self.x*s, self.y*s)
    def __truediv__(self, s):
        float_s=float(s)
        return Punto2D(self.x/float_s, self.y/float_s)
    def __floordiv__(self, s):
        return Punto2D(self.x//s, self.y//s)
    def __repr__(self):
        return 'Punto2D: (%f, %f)'%(self.x, self.y)
```

```
# programa principal----
a=Punto2D(3,5)
b=Punto2D(2,7)
print(a+b)
print(b-a)
print(b*1.3)
print(a//17)
print(a/17)
```