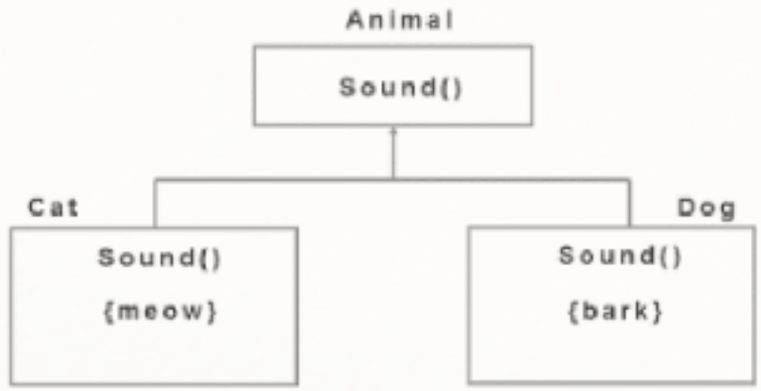
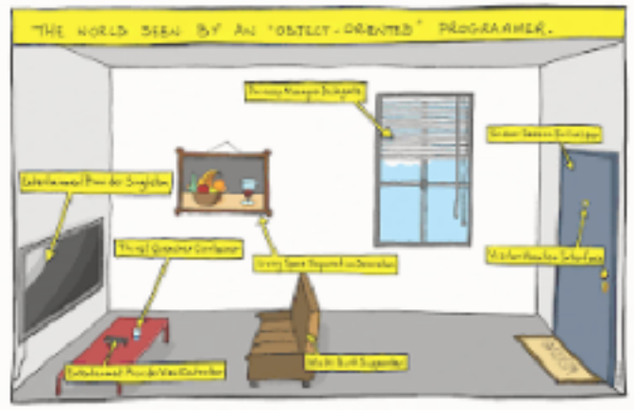
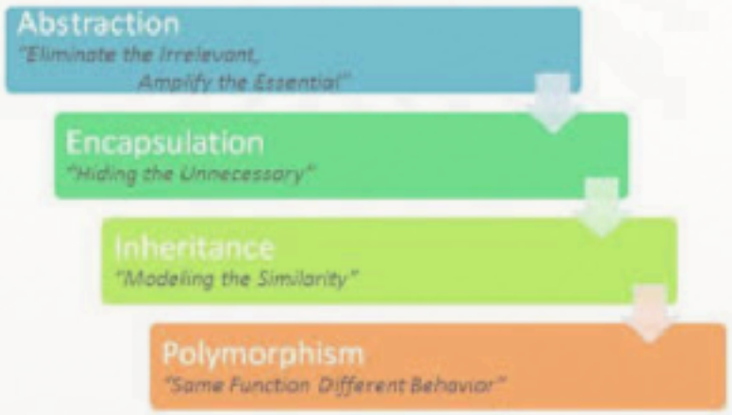
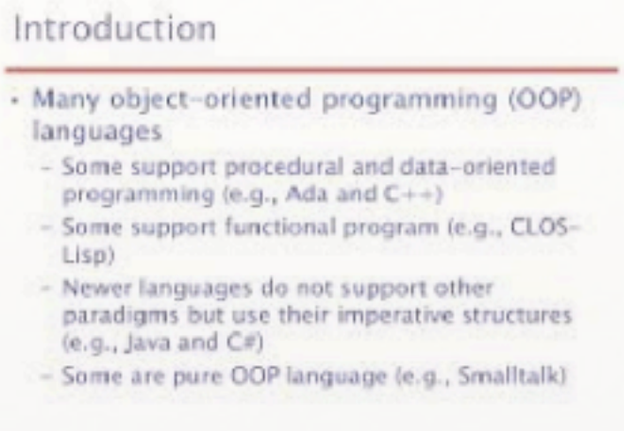








Técnico Especialista em Gestão de Redes e Sistemas Informáticos

Introdução à Programação

UFCD(s): 5118, 5119

Programação Orientada por Objectos em Python

Objectos

- Python suporta vários tipos de dados:

```
987    "Bom dia"  2.7182   True {'um': 1, 'dois': 2}   [20, -1, 78]
```

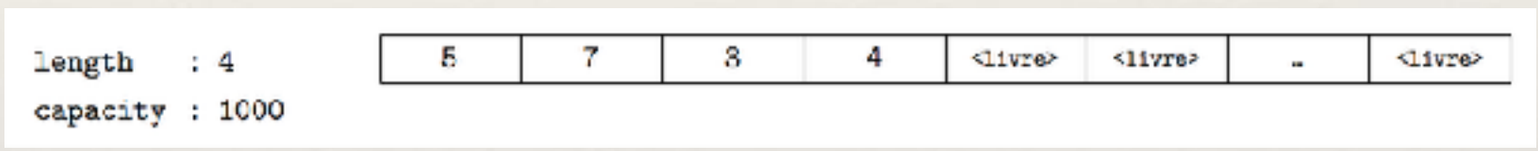
- Cada um dos valores é um **objecto**, e cada objecto tem:
  - um **tipo** de dados
  - uma **representação interna** de dados
  - uma **interface** constituída por um conjunto de funções ou procedimentos
- Um objecto é uma instância de um tipo de dados/classe:
  - `987` é uma instância de **`int`**
  - `"Bom dia"` é uma instância de **`str`**

Objectos

- Objectos são uma **abstração de dados** com:
  - **atributos de dados:** representação interna
  - **métodos**: funções ou procedimentos que operam sobre os dados e que definem a interface do objecto
- Métodos definem comportamento e permitem esconder a representação
- Representação interna deve ser **privada**: acesso directo prejudica futuras alterações e cria dependências

Exemplo

- Como é que uma lista é representada internamente?
  - lista ligada:
  - *array*:



- Operações para implementar listas:
  - `L[i], L[i:k], +, +=`
  - `len(), sum(), min(), max(), del L[i]`
  - `L.append(), L.extend(), L.index(), L.remove(), etc...`
- Estas operações não devem depender da representação interna da lista

POO: Programação Orientada p/ Objectos

- **Encapsulamento:** uma só "unidade" c/ dados e operações
- **Ocultação da Informação**: dados são acessíveis apenas através das operações; não é dado acesso directo aos dados
- **Operações bem definidas**: existem "protocolos" bem definidos para construir objectos, eliminar objectos, aceder a partes dos objectos, modificar essas partes, inspeccionar objectos, etc.
- **Tipo de dados**: objectos pertencem a um tipo de dados integrado com os tipos de dados da linguagem (na maioria dos casos isto envolve classes de objectos - ver à frente)

POO

- **Hierarquias**: é possível estabelecer uma hierarquia de tipos de dados de objectos (na maioria dos casos isto envolve classes de objectos - ver à frente)
- **Herança**: a possibilidade de objectos herdarem dados e comportamento dos tipos de dados na sua hierarquia, e não apenas do seu tipo de dados mais concreto
- **Polimorfismo**(*): no contexto da POO, um objecto pode assumir qualquer um dos tipos de dados definidos na sua hierarquia

(*) - Existem outras formas de polimorfismo: sobrecarga ou funcional(overloading), paramétrico, etc.

POO: Objectivos/Benefícios

- **Decomposição:**
  - Objectos são abstracções (*): divisão de um problema em partes, permite implementar e testar as partes isoladamente
  - diminui complexidade

  (*) - Na maioria das linguagens são definidos a partir de classes, sendo este o mecanismo que permite definir essas abstracções

- **Modularidade**:
  - objecto é um ambiente isolado
  - evita conflitos de nomes

POO: Objectivos/Benefícios

- **Reutilização:**
  - modularidade torna mais fácil partilhar código
  - herança de classes permite que subclasses herdem (ou seja, reutilizem) código das superclasses
    - também podem estender e/ou redefinir comportamentos definidos em superclasses
  - polimorfismo permite que objectos relacionados sejam processados de forma uniforme
- **Manutenção**:
  - polimorfismo permite que objectos possam vir a ser utilizados em situações não previstas inicialmente
  - também permite que classes inteiras sejam substituídas por outras sem necessidade a alterar todo o código

Classe

- Mecanismo a partir do qual criamos objectos: uma classe é como que um **modelo**, um *template*
- Permitem juntar dados - **atributos** - com funcionalidade - **métodos**.
- Representa um **conceito** do domínio da aplicação, ou seja, uma **abstracção**
- Uma classe é um **novo tipo de dados**
- Em Python, classes são **dinâmicas**:
  - criadas em tempo de execução
  - podem ser modificadas com o programa a correr

Definir e Utilizar Classe

- **Definição:**
  - atribuir um **nome**
  - definir os atributos que cada objecto terá
  - ~~exemplo: implementar uma classe para listas~~
- **Utilização:** *da classe*
  - criar novas instâncias de objectos
  - invocar operações sobre os objectos
  - ~~exemplo: criar uma lista com `L = [1, 2]` e aceder ao seu tamanho com `len(L)`~~

Duas Perspectivas

Implementação	Utilização
Definir a classe, dando-lhe um nome; classe é um tipo de dados	Criar objectos (instâncias) da classe através do nome da classe
Definir atributos de dados: do QUE é feito cada objecto	Executar operações - métodos - sobre os objectos
Definir métodos: COMO os objectos devem ser usados	Classe é um tipo de dados => é possível utilizar operações comuns a todos os tipos de dados
Definir COMO os objectos são CRIADOS. Pode ser necessário definir COMO são DESTRUÍDOS	Em determinadas linguagens pode ser necessário garantir que objectos são destruídos

Classe é um Tipo de Dados

- Utilizamos a palavra-reservada **`class`** para definir um novo tipo de dados:

```
class Ponto2D(object): # object opcional em Python 3, recomendado em Python 2
    # seguem-se os atributos "indentados" tal como com def
    # ...
```

- `object` é uma classe da qual todas as classes em Python 3 derivam:
  - `Ponto2D` vai **herdar** todos os atributos de `object` (falamos de herança mais à frente)
  - `Ponto2D` é uma **subclasse** de `object`; esta é a **superclasse** de `Ponto2D`
  - Um objecto da classe `Ponto2D` pode ser utilizado onde se espera um objecto do tipo `object`

Atributos

- Atributos de dados → **Atributos** ou **Campos**
  - objectos que compõem cada objecto desta classe
  - são variáveis de cada instância/objecto
  - exemplo: um `Ponto2D` possui dois números, `x` e `y`
- Atributos de procedimentais → **Métodos**
  - podemos olhar para métodos como funções especializadas nos objectos desta classe
  - acedemos e manipulamos os atributos (de dados) através destes métodos
  - exemplo: distancia_origem é um método que indica a distância para a origem (através da expressão `math.sqrt(x**2 + y**2)`)

Instanciar uma Classe

- **Instanciar classe** ⇔ **Criar objectos** a partir de uma classe
- Necessário definir o método especial (*) **`__init__`**

  (*) - "mágico" ou *dundermethod* no jargão do Python:
- Este método - designado de **construtor** - cria e inicializa os atributos :

```
class Ponto2D:
    def __init__(self, x: float, y: float):
        self.x = x
        self.y = y
```

`self` : parâmetro que representa objecto que está a ser construído
`x` e **`y`**: parâmetros com os dados de inicialização dos atributos
`self.x` e **`self.y`**: atributos

Instanciar uma Classe

```
p = Ponto2D(1, 2)
origem = Ponto2D(0, 0)
print(p.x)
print(f"(X, Y)->({p.x},{p.y})")
```

`Ponto2D(1, 2)`: cria novo objecto do tipo `Ponto2D` e passa 1 e 2 para o `__init__`
`p.x`: acedemos a um atributo através do operador de acesso . (ponto)

- **Atributos/Campos** de dados são também designados de de **variáveis de instância** ou **variáveis membro** (terminologia de C++)
- A classe pode ser vista com uma função construtora que delega trabalho para o `__init__`
- Não é necessário dar um argumento para `self`; o Python trata disso automaticamente

Métodos

- **Função/procedimento** especializado nesta classe
- Python passa sempre o objecto como primeiro argumento do método
  - Convenção é dar o nome `self` ao parâmetro correspondente
- O operador de acesso "." (ponto) é utilizado para aceder a um atributo, de dados ou procedimental (método)

Definir um Método

```
class Ponto2D:
    # ... __init__ ...
    def distancia(self, outro: Ponto2d) -> float:
        x_dist_sq = (self.x - outro.x)**2
        y_dist_sq = (self.y - outro.y)**2
        return math.sqrt(x_dist_sq + y_dist_sq)
```

`self`: o objecto sobre o qual o método é invocado

- Invocação:

```
p1, p2 = Ponto2D(1, 2), Ponto2D(5, -8.6)
dist = p1.distancia(p2)
```

`p1.distancia(p2)` é equivalente a **`Ponto2D.distancia(p1, p2)`**

- Além do **`self`**, métodos são funções regulares: possuem parâmetros, fazem operações, devolvem objectos, podem ser decorados, etc.

Representação Externa de Objecto

- Representação externa pouco informativa dos objectos por omissão

```
>>> Ponto2D(1, 2)
<__main__.Ponto2D object at 0x2eb278516414>
>>> print(Ponto2D(1, 2))
<__main__.Ponto2D object at 0x2eb278516414>
```

- Definimos método **`__str__`** para representação "legível"
- Definimos método **`__repr__`** para representação "técnica" e/ou que permita reconstruir o objecto

Representação Externa de Objecto

```
class Ponto2D:
    # ... __init__ ...
    def __str__(self):
        return f'<{self.x},{self.y}>'
    def __repr__(self):
        return f'Ponto2D({self.x}, {self.y})'
```

```
>>> Ponto2D(1, 2)
Ponto2D(1, 2)
>>> print(Ponto2D(1, 2))
<1,2>
>>> repr(Ponto2D(1, 2)) + ' - ' + str(Ponto2D(1, 2))
'Ponto2D(1, 2) - <1,2>'
```

Integração com o Sistema de Tipos

- Podemos obter o tipo de dados (ie, a classe) de uma instância:

```
>>> p = Ponto2D(5, 7)
>>> type(p)
<class __main__.Ponto2D>
>>> p.__class__
<class __main__.Ponto2D>
>>> type(Ponto2D), type(type(p))
(<class 'type'>, <class 'type'>)
```

- Podemos "perguntar" se um objecto é de um determinado tipo de dados:

```
>>> isinstance(p, Ponto2D)
True
```

Operações Especiais/Genéricas

- Objectos de uma classe podem ser utilizadas com as seguintes operações: `+`, `-`, `*`, `/`, `**`, `<`, `<=`, `>`, `>=`, `==`, `!=`, `len()`, `print()`, `int()`, `float()`, *etc*...
- Se forem implementamos os seguintes métodos "mágicos":

```
__add__(self, outro)      ←  self + outro
__sub__(self, outro)      ←  self - outro
__mul__(self, outro)      ←  self * outro
__truediv__(self, outro)  ←  self / outro
__eq__(self, outro)       ←  self == outro
__radd__(self, outro)     ←  outro + self
__len__(self)             ←  len(self)
__float__(self)           ←  float(self)
__str__(self)             ←  str(self)
... etc ...
```

- Ver: https://docs.python.org/3/reference/datamodel.html#basic-customization

Operações Especiais/Genéricas

```
class Ponto2D:
    # ... __init__ ...
    def __eq__(self, p):
        if type(self) is type(p):
            return self.x == p.x and self.y == p.y
        return NotImplemented
    def __add__(self, p) -> Ponto2D:
        return Ponto2D(self.x + p.x, self.y + p.y)

>>> p1, p2 = Ponto2D(1, 2), Ponto2D(4, 1)
>>> p1 + p2
Ponto2D(5, 3)
>>> p1 == p1, p1 == p2
(True, False)
```

Duas Perspectivas (Resumo)

Classe	Objecto/Instância

Classe é um tipo de dados:
`class Ponto2D`
Novo tipo de dados => `Ponto2D`
	Instância é um objecto em concreto que pertence ao tipo de dados da classe:
	`p = Ponto2D(3, 1)`
	`isinstance(p, Ponto2D) => True`

Classe é um modelo para todos os objectos:
• `self` permite "referir" um objecto durante a definição da classe
• `self` é um parâmetro de todos os métodos
	Cada objecto tem a sua "cópia" dos atributos de dados:
	`p1 = Ponto2D(5, 6)`
	`p2 = Ponto2D(3, -3)`
	`p1.x != p2.x and p1.y != p2.y. => True`
	`p1` e `p2` são **objectos diferentes** da **mesma classe**

Classes define atributos e métodos comuns a todos os objectos/instâncias	Objectos possuem a estrutura definida na classe

Ocultação da Informação

- Noutras linguagens podemos marcar atributos como *private* para que não possam ser acedidos directamente
- Passam a ser acedidos através de métodos *getters* e *setters*
- Em Python utilizamos uma convenção para indicar que atributo é privado:
  - Atributo é prefixado com _ mas apenas se é necessário torná-lo privado; exemplo: `saldo` => `_saldo`
  - Utilizamos `@property` para definir *getter* e *setter*

Ocultação da Informação

```
class Ponto2D:
    def __init__(self, x: float, y: float):
        self._x = x
        self._y = y
    @property
    def x(self):
        return self._x
    @x.setter:
    def x(self, novo_x):
        self._x = novo_x

    # e o mesmo para self._y ...
    # ... restantes métodos ...

>>> p1 = Ponto2D(3, -5)
>>> print(p1.x, p1.y)
>>> p1.x = 9
```

self

`self._x` e `self._y`: atributos de dados privados

`property`: permite definir *getter* e *setters* para atributos

Caso se pretenda alterar a forma de armazenamento para, por exemplo, um *array* com dois valores:

```
class Ponto 2D:
    def __init__p(self, x: float, y: float):
        self._par = array.array('d', [x, y])
    @property
    def x(self):
        return self._par[0]
```

Construtores Alternativos

- É conveniente disponibilizarmos construtores alternativos
  - Dados para criar objectos chegam de "fontes" diferentes
  - Por vezes a representação externa dos objectos não coincide com os dados necessários pelo `__init__`
- Métodos de classe - decorados com `@classmethod` - permitem definir construtores alternativos
- Vamos supor que pretendemos um construtor para um `Ponto2D` "codificado" numa string no formato `'<X, Y>'`

Construtores Alternativos

```
class Ponto2D:
    # ... __init__ ...
    @classmethod
    def from_str(cls, ponto: str):
        coords = ponto.strip().split(',')
        return cls(float(coords[0][1:]),
                   float(coords[1][:-1]))
    # ... restantes métodos ...
```

```
>>> p1 = Ponto2D(5, -7)
>>> p2 = Ponto2D.from_str('<5,-7>')
>>> p1 == p2
True
```

`@classmethod`: permite definir um construtor alternativo
`cls`: representa a classe do objecto que ser quer construir; pode variar com herança (de que falaremos mais à frente)

Hierarquias e Derivação de Classes

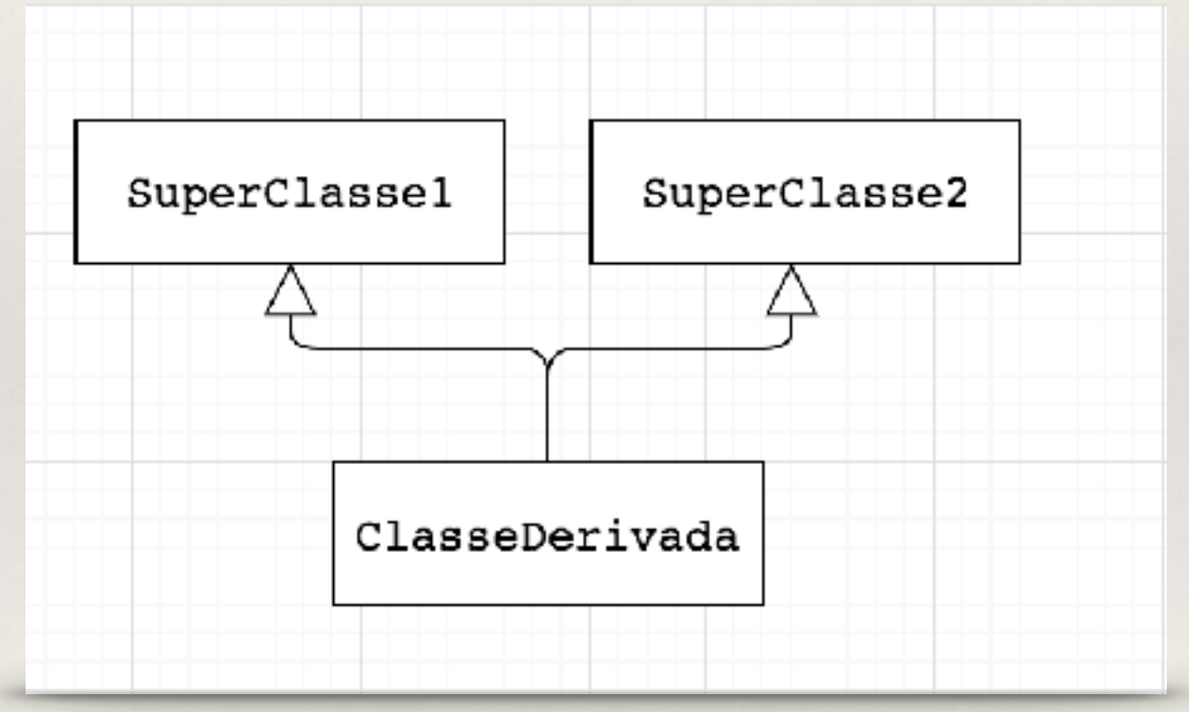
- Python permite definir classe `A` baseada na definição de outra classe `B`:
  - Designamos este mecanismo por **derivação** ou **herança**
  - Classe `B` é designada de **superclasse**, classe de **base** ou classe **genérica**
  - Classe `A` é designada de **subclasse**, classe **derivada** ou classe **especializada**
  - Objectos da classe `A` ou de classes derivadas de `A` são também objectos da classe `B`; o inverso não é verdade
  - Na verdade, em Python, classe `A` pode herdar (ou derivar) da classe `B`, `C`, `D`, etc., ou seja de múltiplas classes => **herança múltipla**
- Herança e derivação são o mecanismo por excelência para **reutilização de código** com classes

Hierarquias e Derivação de Classes

- Subclasses:
  - **Herdam** atributos de dados e comportamento (métodos) das superclasses
  - Podem **estender** superclasses acrescentando outros atributos e métodos
  - Podem reescrever/**redefinir** (*override*) métodos e atributos:
    - Redefinição de um método pode ser completamente diferente ou pode reutilizar método da superclasse; neste caso a redefinição é também uma extensão

Sintaxe

```
class ClasseDerivada(SuperClasse1, SuperClasse2, etc):
    <instr-1>
    .
    .
    <instr-N>
```



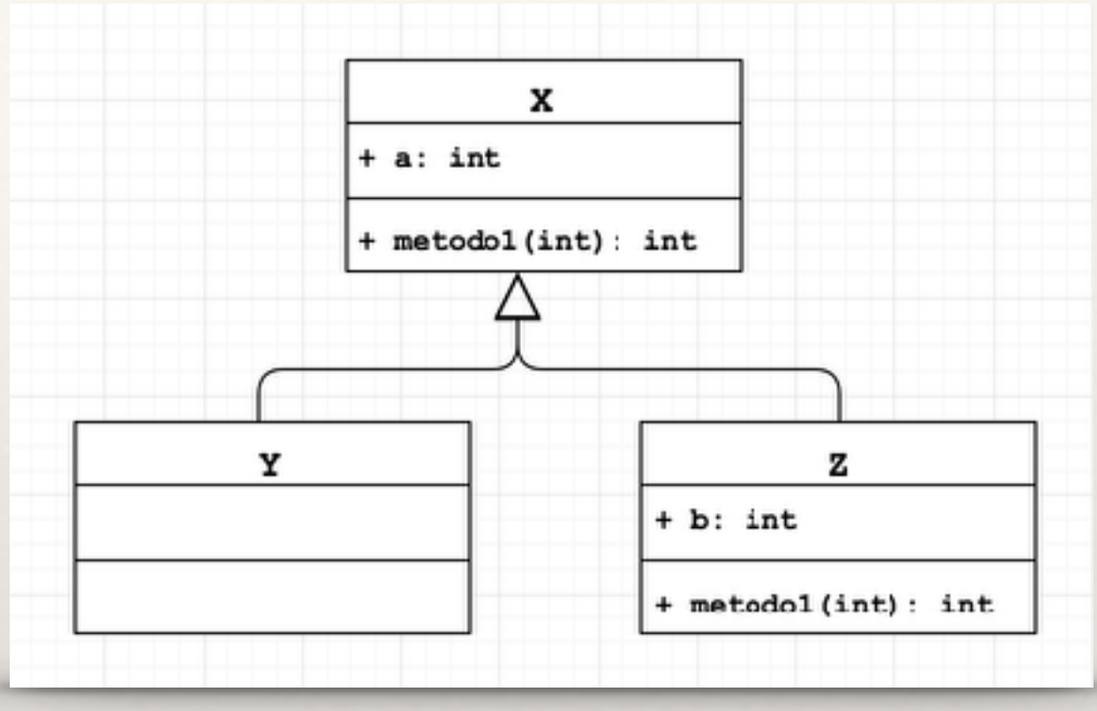
Exemplo 1:

```
class X:
    def __init__(self, a):
        self.a = a
    def metodo1(self, k):
        return self.a + k


class Y(X):
    pass

class Z(X):
    def __init__(self, b, a):
        super().__init__(a)
        self.b = b

    def metodo1(self, k):
        return super().metodo1(k) + self.b + 1
```



Classes `Y` e `Z` herdam atributo `a` e método `metodo1`.

Classe `Z` *estende* classe `X` com atributo `b`.

Classe `Z` *redefine* e *estende* método `X.metodo1`

Função interna `super` devolve objecto para aceder a atributos e métodos das superclasses

Exemplo 1:

```
class X:
    def __init__(self, a):
        self.a = a
    def metodo1(self, b):
        return self.a + b

class Y(X):
    pass

class Z(X):
    def __init__(self, b, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.b = b

    def metodo1(self, b):
        return super().metodo1(b) + self.b + 1
```

```
>>> obj1 = X(10)
>>> obj2 = Y(10)
>>> obj3 = Z(15, 10)
>>> for obj in (obj1, obj2, obj3):
        print(type(obj).__name__,
              obj.metodo1(3))
X 13
Y 13
Z 29
```

`super().__init__(*args, **kwargs)`:permite chamar "superconstrutor" sem ser necessário saber quantos parâmetros são necessários

`Z.__init__` retira para si apenas os parâmetros que são específicos da classe `Z`

Exemplo 2:

Considere uma loja para venda online de produtos. Um **`Produto`** *possui um identificador alfa-numérico, preço e taxa de IVA, valores que deverão ser passados ao construtor* **`__init__`**. *Devem ser aceites os tipos de dados* **`str`** *e* **`decimal.Decimal`** *para parâmetros que representem montantes ou taxas. Deve ser possível obter o montante de IVA e o preço final, sendo que este resulta de acrescentar o montante de IVA ao preço. A representação externa legível* **`__str__`** *deve exibir o tipo de produto, o identificador e o preço. Deve também existir um método* **`to_csv`** *(comma separated values) que "exporta" um produto para uma string com os valores dos atributos delimitados por* `';'`. *Deve ser possível construir um produto a partir desta representação externa através do construtor* `from_csv`. *Os elementos devem ser exportados pela ordem pela qual são passados para o método* `__init__`. *A representação interna dos objectos* **`__repr__`** *deve ser o nome da classe seguido de uma lista com os argumentos a "apresentar" ao* **`__init__`**.

Existem dois tipos de produtos: livro e jogo de computador. Para cada **`Livro`** *são guardados título, código isbn e lista de autores. À excepção de traços ('-'), um isbn deve possuir 10 ou 13 dígitos. Não necessita de considerar outras validações.*

Cada **`JogoComputador`**, *além dos atributos gerais que todos os produtos têm, possui título, número de série e género. O número de série é um inteiro com pelo menos 7 dígitos. Também pode ser uma string com dígitos e traços. O delimitador para a representação csv é o caractere* `':'`.

Exemplo 2:

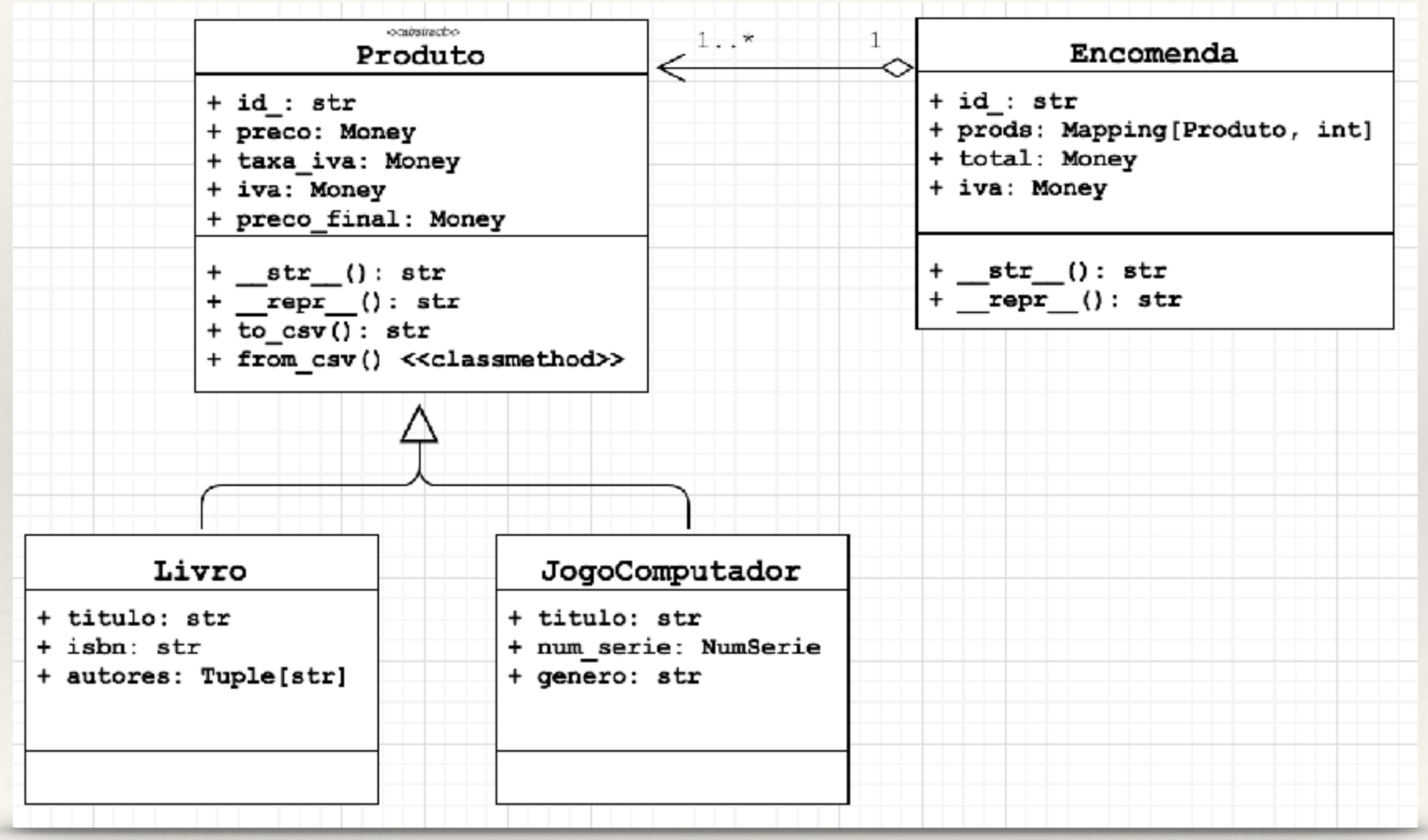
A representação externa e interna dos livros e dos jogos deve ser idêntica à dos produtos gerais. A representação CSV de todos os produtos deve ser o mais parecida possível. Esta deve gerar uma linha de texto na qual os atributos aparecem pela ordem pela qual devem ser passados ao **`__init__`**. *Isto implica que primeiro devem surgir os atributos específicos de cada produto e depois os atributos definidos em* **`Produto`**.

Também deve ser possível representar uma **`Encomenda`**. *Esta consiste de um identificador alfa-numérico, uma associação entre* **`Produto`**s *e quantidades, e uma data-hora de criação. Deve ser possível calcular o preço final de uma encomenda assim como o montante de iva da encomenda.*

Resumindo:

`Produto`	☞	`id: str, preco: str` ou `Decimal, taxa_iva: str` ou `Decimal`
`Livro`	☞	`Produto` com `titulo: str, isbn: str, autores:` lista imutável de nomes
`JogoComputador`	☞	`Produto` com `titulo: str, num_serie: str, genero: str` ou `Decimal`
`Encomenda`	☞	`id: str, prods:` associação `Produto` → quantidade, `data/hora`

Exemplo 2:


«abstract»
Produto
+ id_: str
+ preco: Money
+ taxa_iva: Money
+ iva: Money
+ preco_final: Money
+ __str__(): str
+ __repr__(): str
+ to_csv(): str
+ from_csv() <<classmethod>>
1..*
1
Encomenda
+ id_: str
+ prods: Mapping[Produto, int]
+ total: Money
+ iva: Money
+ __str__(): str
+ __repr__(): str
Livro
+ titulo: str
+ isbn: str
+ autores: Tuple[str]
JogoComputador
+ titulo: str
+ num_serie: NumSerie
+ genero: str

Exemplo 2:

```python
Money = Union[Dec, str]
NumSerie = Union[int, str]

class Produto(ABC):
    csv_delim = ';'

    @abstractmethod
    def __init__(self, id_: str, preco: Money, taxa_iva: Money):
        self.id_ = id_
        self.preco = Dec(preco)
        if self.preco < 0:
            raise ValueError(f'Preço inválido: {preco}')
        self.taxa_iva = Dec(taxa_iva)
        if self.taxa_iva < 0:
            raise ValueError(f'Taxa de IVA inválida: {taxa_iva}')

    @property
    def iva(self) -> Money:
        return self.preco * (self.taxa_iva/100)

    @property
    def preco_final(self) -> Money:
        return self.preco + self.iva
```

`Money e NumSerie`: tipos de dados estáticos; servem para documentar o código e para verificadores estáticos de tipos detectarem erros

`Union`: definido no módulo `typing`; permite indicar que se aceitam vários tipos; em tempo de se execução variáveis são objectos de um desses tipos

`Produto` é uma classe abstracta e daí herdar de `ABC` e o construtor ser decorado com `@abstractmethod`

Por brevidade não indicamos os imports

Exemplo 2:

```python
class Produto(ABC):
    # (continua)

    def __repr__(self):
        class_name = type(self).__name__
        attrs = self.__dict__
        attrs_repr = ',\n'.join(f'    {attr}={val!r}' for attr, val in attrs.items())
        return f'{class_name}(\n{attrs_repr}\n)'

    def __str__(self):
        class_name = type(self).__name__
        return f"{class_name} id: '{self.id_}' preço: {self.preco:.2f} iva: {self.taxa_iva:.2f}%"

    @classmethod
    def from_csv(cls, csv: str):
        delim = cls.csv_delim
        attrs = [literal_eval_extended(val) for val in csv.strip().split(delim)]
        return cls(*attrs)

    def to_csv(self):
        delim = self.csv_delim
        return delim.join(repr(val) for val in self.__dict__.values())
```

`{val!r}`: igual a `{repr(val)}`

`literal_eval`: função do módulo `ast` que opera o inverso da função `repr`, ou seja, dada uma string com a representação interna/técnica de um objecto, constrói esse objecto; limitação: não suporta `datetime` nem `Decimal`.

`literal_eval_extd.`: versão de `literal_eval` que suporta `datetime` e `Decimal`.

`__dict__` : dicionário que cada objecto tem com a lista de atributos; em Python >= 3.6 mantém a ordem de inserção

Exemplo 2:

```python
class Livro(Produto):
    def __init__(self, titulo: str, isbn: str, autores: Tuple[str], *args, **kwargs):
        isbn_sem_tracos = isbn.replace('-', '')
        if not isbn_sem_tracos.isdigit() or len(isbn_sem_tracos) not in (10, 13):
            raise ValueError(f'ISBN inválido: {isbn}')
        self.titulo = titulo
        self.isbn = isbn
        self.autores = autores
        super().__init__(*args, **kwargs)


class JogoComputador(Produto):
    csv_delim = ':'

    def __init__(self, titulo: str, num_serie: NumSerie, genero: str, *args, **kwargs):
        num_serie_sem_tracos = str(num_serie.replace('-', ''))
        if not num_serie_sem_tracos.isdigit() or len(str(num_serie_sem_tracos)) < 7:
            raise ValueError(f'Número de série inválido: {num_serie}')
        self.titulo = titulo
        self.num_serie = int(num_serie_sem_tracos)
        self.genero = genero
        super().__init__(*args, **kwargs)
```

Classes apenas necessitam de definir construtor; todos os atributos e comportamento definidos em `Produto` são herdados

`Tuple[str]`: tuplo de strings

`csv_delim`: variável de classe definida em `Produto` é redefinida em `JogoComputador`; métodos `Produto.from_csv/to_csv` não necessitam de ser alterados

Exemplo 2:

```python
class Encomenda:
    def __init__(self, id_: str, prods: Dict[Produto, int]):
        self.id_ = id_
        self.data_hora = datetime.now()
        self.prods = prods

    @property
    def total(self) -> Money:
        return sum(prod.preco_final * quant for prod, quant in self.prods.items())

    @property
    def iva(self) -> Money:
        return sum(prod.iva for prod in self.prods)

    def __repr__(self):
        class_name = type(self).__name__
        prods = ', '.join(f'{prod.id_!r}: {quant}' for prod, quant in self.prods.items())
        return f'{class_name}({self.id_!r}, {self.data_hora!r}, #[{prods}])'
```

Exemplo 2:

```
liv1 = Livro(
    titulo='Automate the Boring Stuff with Python',
    isbn='978-1593275990',
    autores=('Al Sweigart',),
    id_='LL12',
    preco='20',
    taxa_iva='13',
)
liv2 = Livro.from_csv("'Python Cookbook';'978-1-449-34037-7';('David Beazley', 'Brian K. Jones');'LL98';'30';'13'")
liv3 = Livro.from_csv(liv2.to_csv())

jog1 = JogoComputador(
    titulo='Assassins Creed IV',
    num_serie='8561245220',
    genero='FPS',
    id_='JC11',
    preco='100',
    taxa_iva='13',
)
```

Exemplo 2:

```
jog2 = JogoComputador.from_csv("'Counter-Strike':'8111135910':'FPS':'JC20':Decimal('98'):Decimal('13')")

print('-----')
for p in (liv1, liv2, jog1, jog2):
    print(repr(p))
    print(p)
    print('preço final:', p.preco_final)
    print('-----')

enc1 = Encomenda(id_='12PQ78', prods={liv1: 2, liv2: 1, liv3: 3, jog1: 1, jog2: 1})
print(enc1.total)
print(enc1.iva)
```

Referências

[1] *"9. Classes - The Python Tutorial"*, https://docs.python.org/3/tutorial/classes.html

[2] João Pavão Martins, *"Programação em Python: Introdução à Programação Utilizando Múltiplos Paradigmas, 2a Edição"*, IST Press, 2017, Cap. 11

[3] Guttag, John, *"Introduction to Computation and Programming Using Python - Revised and Expanded Edition"*, MIT Press, 2013, Cap. 11

[4] Raymond Hettinger, *"Python's Class Development Toolkit"*, https://www.youtube.com/watch?v=HTLu2DFOdTg , YouTube Next Day Video