INTRODUÇÃO À PROGRAMAÇÃO

UFCD(s) 5118, 5119

GUIA DE LABORATÓRIO 4.1
CLASSES E PROGRAMAÇÃO ORIENTADA POR OBJECTOS (ßeta)

OBJECTIVOS

- Introduzir/Aprofundar os seguintes conceitos: **Classes**, **Objectos**, **Métodos**, **Atributos**, **Construtores**
- Aprender a utilizar classes e métodos para organizar programas
- Introduzir a Programação Orientada por Objectos

INSTRUÇÕES

Classes e Objectos

1. Inicie o REPL do Python e abra o editor de texto/IDE que costuma utilizar.

2. Crie o ficheiro de código `classes1.py` e importe (para já) o tipo `date` definido em `datetime`.

*Uma **classe** é uma colecção de variáveis, propriedades e métodos agrupados sob um único nome. De um modo geral, este agrupamento reflecte um **conceito** do domínio para o qual a aplicação foi desenhada, conceito esse que pretendemos representar em Python (ou noutra linguagem que suporte classes). No âmbito da **Programação Orientada Por Objectos** (**POO** ou apenas **OO**), uma classe define um tipo de dados de objectos do mundo real. No fundo, uma classe representa um **modelo** ou **especificação** da implementação desses objectos. Este modelo descreve que estados e comportamentos esses objectos vão exibir.*
*Os **objectos** são **instâncias** desse padrão ou modelo, isto é, são representações "vivas" do modelo. A classe contém a especificação dos objectos, e cada um destes é criado a partir da especificação dada pela classe. Quando um objecto é criado a partir de uma classe, dizemos que esse objecto pertence a um **tipo de dados** cujo nome é o nome da classe. Ou seja, podemos declarar variáveis pertencentes ao tipo introduzido pela declaração de uma classe e o compilador fará as verificações semânticas habituais sobre variáveis deste novo tipo.*
*Usando terminologia de POO, uma classe reúne **atributos** e **comportamento**. Os atributos, ou **campos**, são as **variáveis internas** e **propriedades**, definidas dentro da classe. Por seu turno, o conjunto de **métodos** ou funções definidos dentro da classe é aquilo a que se designa pelo comportamento da classe, ainda que também se possam designar por operações ou atributos comportamentais.*
*A noção de classe é similar à nocão de **Tipo de Dados Abstracto** (TDA). Realmente, um TDA corresponde a um conjunto de "valores" ou objectos e às operações disponíveis para esses objectos. Ao conjunto dessas operações designamos **interface** entre o TDA e o "resto do mundo". As classes são o mecanismo por excelência que o Python oferece para implementar TDAs. Outras linguagens, como Java e C#, acrescentam ainda outros mecanismos como interfaces e classes abstractas.*

3. Vamos supor que pretendemos gerir um catálogo de livros de uma biblioteca. Neste âmbito, queremos representar um livro com os seguintes atributos: código ISBN, titulo, autores, género, data de criação, número de exemplares e tipo de suporte. Ao ficheiro `classes1.py` vamos adicionar a classe `Livro`

```
class Livro:
```

```
def __init__(self):
   self.cod_ISBN = None
   self.titulo = 'N/D'
   self.genero = 'N/D'
   self.autores = []
   self.data = None
   self.num_exemplares = -1
   self.suporte = 'N/D'

def __str__(self):
   return "{}: {} {}".format(self.titulo, ','.join(self.autores), self.data)
```

*Definimos classes com a palavra-reservada **`class`**. Ao contrário de outras linguagens, como Java ou C#, os campos ou atributos de cada objecto são acrescentados dinâmicamente ao objecto que será criado através de um método especial, o método **`__init__`** (ver em baixo).*
*Opcionalmente, a seguir ao nome da classe podemos indicar uma lista de classes a partir da qual esta classe deriva (ie, herda). Por exemplo, se tivéssemos escrito `class Livro(Documento)`, então a classe `Livro` herdaria da classe `Documento`. Em Python 3, todas as classes herdam da classe **`object`** quando não indicamos nada.*
*Uma classe serve para criar objectos "baseados" na especificação da classe. O argumento `self` do método **`__init__`** representa o próprio objecto que está a ser criado.*

A definição de uma classe pode incluir os seguintes elementos (*):
*- **Campos**: Por vezes designados por **variáveis-membro** ou **atributos**, os campos são variáveis com informação respeitante a cada objecto. Se um campo for definido dentro de um método, como é o caso desta classe, onde os campos são definidos no método **`__init__`**, então cada objecto possui a sua cópia desse campo.*
*- **Variáveis de Classe**: São variáveis definidas dentro da classe, fora de qualquer método. São partilhadas por todos os objectos.*
*- **Métodos**: Métodos são funções definidas dentro de uma classe. Eles representam o comportamento dos objectos, isto é, as operações que os objectos conseguem desempenhar. Os métodos levam a cabo a lógica de programação que está por trás dos algoritmos e dos acessos aos dados. O primeiro parâmetro de um método é sempre o objecto sobre o qual o método é invocado, representado normalmente pelo nome `self`. Alguns métodos, que não recebem o objecto como primeiro argumento, são métodos partilhados por todos os objectos e são designados por métodos estáticos. Outros recebem como primeiro parâmetro a própria classe e são designados métodos de classe. Veremos à frente para que servem estes métodos e como podem ser definidos. Alguns métodos são especiais e indicamos isso prefixando e sufixando-os com dois 'underscores'. A nossa classe define dois desses métodos o método **`__init__`** e o método **`__str__`** . Este segundo método permite obter uma representação "legível" do objecto.*
*- **Propriedades**: As propriedades são métodos especiais - designados de **acessores** - para aceder a determinados campos, sendo que elas próprias são utilizadas como se fossem campos. Uma propriedade implementa os métodos **`get`** e **`set`**.*
*- **Construtores/Inicializadores**: Também são métodos, e são invocados quando criamos um objecto da classe. Em Python os construtores são os métodos **`__init__`** e **`__new__`** (é raro definirmos este último). Alguns autores não classificam o método **`__init__`** de construtor, dizem apenas que é um inicializador. Aqui continuaremos a designar **`__init__`** por construtor.*
*- **Outros**: Em Python, uma classe pode conter outras classes e até código arbitrário como **`print`**s e ciclos **`for`**. Uma classe é muito similar a um módulo.*
*(*) - As classes também podem incluir a definição de **destrutores**, tópico que não vamos abordar.*

*Uma classe suporta dois tipos de operações básicas: **instanciação** para criar um objecto, e acesso a atributos (campos ou métodos) através da **notação ponto** "dot notation". Por exemplo, `Livro()` é a operação de instanciação que permite criar um objecto da classe `Livro`. Por seu turno, para acedermos ao atributo `titulo` de um objecto `Livro` qualquer fazemos `obj.titulo` .*

4. Agora podemos utilizar objectos da classe `Livro` da seguinte forma:

```
liv1 = Livro()
liv1.cod_ISBN = '978-1449320416'
liv1.titulo = 'Programming C# 5.0'
liv1.genero = 'Informática'
liv1.autores = ['Ian Griffiths']
liv1.data = date(2012, 10, 31)
liv1.num_exemplares = 20
liv1.suporte = 'Capa Mole'

liv2 = Livro()
liv2.cod_ISBN = '978-0321563842'
liv2.titulo = 'The C++ Programming,'\
              'Language 4th Edition'
liv2.genero = 'Informática'
liv2.autores = ['Bjarne Stroustrup']
liv2.data = date(2013, 5, 19)
liv2.num_exemplares = 17
liv2.suporte = 'Capa Mole'
```

*Instanciamos objectos de uma classe fazendo `Classe(<argumentos para os parâmetros de __init__>)`. A operação de instaciação aloja em memória espaço para um objecto (invocando, possivelmente, o método **`__new__`**) e invoca o método **`__init__`** para inicializar o objecto. Exemplo:*

```
Livro liv = Livro()
```

O objecto referenciado pela variável `liv` é do tipo de dados `Livro` e permite-nos aceder ao objecto devolvido pela operação de instanciação. Por vezes mencionamos o objecto através do nome da variável: "o objecto `liv`".
*Um **construtor** é um método que nos permite, precisamente, controlar a inicialização dos objectos. Podemos definir quantos construtores quisermos, mas apenas pode existe um método **`__init__`**. Para tal, devemos definir métodos de classe, algor que veremos mais à frente.*

Em C#, a classe anterior poderia ser criada da seguinte forma:

```
class Livro {
   public string codISBN;
   public string titulo;
   public string genero;
   public string[] autores;
   public DateTime data;
   public uint numExemplares;
   public string suporte;
}
```

*Os atributos são especificados fora de qualquer método. Além disso, note-se que necessitamos de indicar o tipo de dados de todos os atributos, bem como um modificador de acesso - **`public`**, neste caso - que indica se é permitido o acesso ao atributo ou não. Nessas linguagens, podemos indicar que determinados campos apenas podem ser acedidos por métodos definidos dentro da classe "marcando-os" com modificadores como **`private`** ou **`protected`**. Um dos conceitos fundamentais da Programação Orientada por Objectos (POO) é o conceito de encapsulamento. **Encapsulamento** consiste em empacotar atributos de dados (os campos) com atributos comportamentais (os métodos) num só objecto. Associado a esse conceito temos um outro, **ocultação de informação** (data/information hiding), que passa por restringir o acesso à representação interna dos objectos. Porquê? Para que possamos modificar essa representação interna sem alterar o código que utiliza a classe. Ocultação da informação pode ser obtida através dos tais modificadores de acesso. A linguagem Python não possui modificadores de acesso, mas dispõe de **propriedades** que são métodos especiais que podem ser alterados sem que o código cliente da classe tenha que ser alterado. Por outro lado, em Python dispomos de iteradores, geradores, funções de primeiro nível e closures, "mecanismos" que, indirectamente, facilitam o encapsulamento e providenciam ocultação de informação.*

5. Agora aceda ao REPL e experimente:

```
>>> liv1
<__main__.Livro object at 0x1022eb588>
>>> print(liv1)
```

```
Programming C# 5.0: Ian Griffiths 2012-10-31
>>> liv1.num_exemplares + 1
21
```

6. Queremos controlar a inicialização dos objectos da classe `Livro`. Por exemplo, determinados atributos, como o título, o género e os autores, devem ser passados logo na construção do objecto. Para os restantes atributos, queremos definir valores por omissão "apropriados". Vamos acrescentar parâmetros ao método **`__init__`** :

```
class Livro:
    def __init__(self, titulo, genero, autores):
        self.titulo  = titulo
        self.genero  = genero
        self.autores = autores

        self.cod_ISBN       = None
        self.data          = date.today()  # data por omissão
        self.num_exemplares = 0
        self.suporte       = 'Capa Dura'
```

*Dentro da classe (ie, no código de um método, propriedade ou construtor), utilizamos o **`self`** e o ponto para aceder aos campos do objecto a ser criado. Este parâmetro, que é similar à palavra-reservada **`this`** das linguagens derivadas de C++ (como Java e C#), representa o próprio objecto sobre o qual o método é chamado. Consideremos, por exemplo, o seguinte código:*

```
txt = 'A_B_C'
txt.split('_')
```

*Dentro da classe **`str`** (tipo de dados do objecto `txt`), o **`self`** representa o objecto com os dados que esses métodos manipulam. Quando o método `str.split` for executado, o **`self`** vai representar a string que está à esquerda do ponto na altura da invocação, ou seja, neste exemplo o **`self`** dentro do método `split` vai ser a string `txt`.*

*Quando não definimos um construtor, a linguagem Python acrescenta à definição da classe um construtor vazio, ie, um construtor sem parâmetros - para além de **`self`** - e sem instruções (eg: **`def __init__(self): pass`**). Porém, após ter sido definido um construtor, somos obrigados a utilizar esse construtor.*

Por exemplo, dada a seguinte classe:

```
class Xpto:
  def __init__(self, a): self.a = a
```

… somos obrigados a inicializar objectos da classe `Xpto` da seguinte forma: `obj = Xpto(<exp. inteira>)`

7. Agora pode alterar o código do em `classes1.py` que cria os dois objectos para:

```
liv1 = Livro('Programming C# 5.0', 'Informática', ['Ian Griffiths'])
liv1.cod_ISBN = '978-1449320416'
liv1.data = date(2012, 10, 31)
liv1.num_exemplares = 20
liv1.suporte = 'Capa Mole'

liv2 = Livro('The C++ Programming Language, 4th Edition', 'Informática',
             ['Bjarne Stroustrup'])
liv2.codISBN = '978-0321563842'
liv2.data = date(2013, 5, 19)
```

```
liv2.num_exemplares = 17
liv2.suporte = 'Capa Mole'
```

8. Agora aceda ao REPL e tente:

```
>>> print(liv1)
Programming C# 5.0: Ian Griffiths 2012-10-31
>>> liv1.autores
['Ian Griffiths']
```

9. Os valores por omissão dos campos sugerem a utilização de parâmetros por omissão:

```
class Livro:
    def __init__(self, titulo, genero, autores,
                 cod_ISBN=None, data=None, num_exemplares=0,
                 suporte='Capa Dura'):
        self.titulo  = titulo
        self.genero  = genero
        self.autores = autores
        self.cod_ISBN      = cod_ISBN
        self.data          = data if data else date.today() # data por omissão
        self.num_exemplares = num_exemplares
        self.suporte       = suporte
```

10. Por seu turno, quando temos uma função com tantos parâmetros é melhor utilizar argumentos com nome, mesmo para os parâmetros obrigatórios:

```
liv1 = Livro(
    titulo='Programming C# 5.0',
    genero='Informática',
    autores=['Ian Griffiths'],
    cod_ISBN = '978-1449320416',
    data = date(2012, 10, 31),
    num_exemplares = 20,
    suporte = 'Capa Mole'
)

liv2 = Livro(
    titulo='The C++ Programming Language, 4th Edition',
    genero='Informática',
    autores=['Bjarne Stroustrup'],
    cod_ISBN='978-0321563842',
    data = date(2013, 5, 19),
    num_exemplares = 17,
    suporte = 'Capa Mole'
)
```

11. E agora definimos mais um livro:

```
liv3 = Livro(
    titulo="Manual Tipográfico",
    genero="Tipografia",
    autores=['N/D']
)
```

12. Vamos também acrescentar o método `mostra` à classe Livro que nos permitirá "inspeccionar" o conteúdo de um `Livro` em maior detalhe:

```
class Livro:
    def mostra(self):
        print('-' * 80)
        print("LIVRO: ", self.titulo)
        print('-' * 80)
        print("{:<25}: {}".format(
              "Código ISBN",
               self.cod_ISBN if self.cod_ISBN else "N/D"
        ))
        print("{:<25}: {}".format("Género", self.genero))

        legenda = "Autores"
        for autor in self.autores:
            print("{:<25}: {}".format(legenda, autor))
            legenda = ""

        print("{:<25}: {}".format("Data", self.data))
        print("{:<25}: {}".format("Número de Exemplares", self.num_exemplares))
        print("{:<25}: {}".format("Suporte", self.suporte))
```

Podemos invocar o método mostra de duas formas:

1. `Livro.mostra(liv1)`
2. `liv1.mostra()`

Naturalmente que a 2a forma é mais conveniente. De facto, quando utilizamos a 2a forma, o Python sabe que deve utilizar como argumento parâmetro `self` o objecto que está à esquerda do ponto.

13. Teste no REPL:

```
>>> liv1.mostra()
--------------------------------------------------------------------------------
LIVRO:  Programming C# 5.0
--------------------------------------------------------------------------------
Código ISBN              : 978-1449320416
Género                   : Informática
Autores                  : Ian Griffiths
Data                     : 2012-10-31
Número de Exemplares     : 20
Suporte                  : Capa Mole

>>> liv2.mostra()

--------------------------------------------------------------------------------
LIVRO:  The C++ Programming Language, 4th Edition
--------------------------------------------------------------------------------
Código ISBN              : 978-0321563842
```

```
Género                  : Informática
Autores                 : Bjarne Stroustrup
Data                    : 2013-05-19
Número de Exemplares    : 17
Suporte                 : Capa Mole
```

14. Podemos definir outro construtor para receber uma string de atributos separados por vírgulas. Esse construtor não se pode chamar **`__init__`** e deve ser prefixado com **`@classmethod`**. Um convenção habitualmente utilizada passa por utilizar o nome `from_qualquercoisa` (ou `fromQualquerCoisa`):

```
class Livro:
    @classmethod
    def from_string(cls, txt):
        attrs = txt.split(',')
        return cls(
            titulo=attrs[0],
            genero=attrs[1],
            autores=[autor.strip() for autor in attrs[2].split('/') if autor],
            cod_ISBN=attrs[3],
            data=datetime.strptime(attrs[4].strip(), '%Y-%m-%d').date(),
            num_exemplares=int(attrs[5]),
            suporte=attrs[6],
        )
```

> *O decorador **`@classmethod`** indica que o método deve receber como primeiro argumento, não o objecto que está a ser criado, mas a própria classe. Porquê receber a própria classe? Devido ao mecanismo de herança, conforme veremos à frente. Note-se que ao fazermos `cls(titutlo=attrs[0], … etc.)` estamos a construir o objecto e a invocar **`__init__`**.*

15. Este construtor pode agora ser utilizado da seguinte forma:

```
>>> txt = 'Starting Out With Python 2nd Ed.,Informática,Tony Gaddis/,978-0-13-257637-6,2012-10-31,20,Capa Mole'
>>> liv4 = Livro.from_string(txt)
<__main__.Livro object at 0x1019ba710>
>>> liv4.cod_ISBN, liv4.suporte
('978-0-13-257637-6', 'Capa Mole')
```

16. Agora podemos utilizar o construtor com mais parâmetros para validar determinados parâmetros. Por exemplo:

- Evitar que o titulo seja vazio (`None`, `''` ou uma colecção vazia de elementos)
- Evitar que o género seja vazio e verificar se pertence a uma lista pré-definida de géneros
- Evitar que a lista de autores esteja vazia
- Evitar que o código ISBN seja inválido. Verificar um ISBN integralmente é algo complicado, mas podemos fazer uma verificação mínima: testar se tem 13 dígitos.
- Evitar que o suporte não seja um dos suportes pré-definidos

Vamos desenvolver algumas funções internas na classe `Livro` para efectuar a validação destes elementos.

Comece por acrescentar a função `codISBNValido`:

```
class Livro:
    # Construtores e mostra ...
    @staticmethod
    def cod_ISBN_valido(cod_ISBN):
        cod_ISBN = cod_ISBN.replace('-', '')
        if len(cod_ISBN) < 13:
            return False
        return cod_ISBN.isdigit()
```

*Esta função retorna **`true`** se o código ISBN passado como argumento for um ISBN válido. Note-se que a função não aceita que o ISBN esteja vazio. No entanto, uma vez que nem todos os `Livros` possuem ISBN, o construtor apenas invoca esta função para um ISBN não vazio.*
*Como não pretendamos utilizar este método (note-se que no parágrafo anterior utilizámos a designação 'função' e não método; não foi despropositado) noutro contexto, definimo-lo dentro da classe `Livro` apesar de ele não "receber" um objecto do tipo `Livro`. Caso contrário, poderia ser uma função definida fora da classe. Por este motivo, prefixámos o método com o decorador **`@staticmethod`**. Agora podemos invocar o método com `Livro1.cod_ISBN_valido('xpto')` ou com `liv1.cod_ISBN_valido('xpto')`.*

17. Agora vamos alterar a definição do **`__init__`** de modo a incluir uma chamada a esta função. Se o código ISBN não for válido, este construtor lança uma excepção do tipo `ValueError`:

```
class Livro:
    def __init__(self, titulo, genero, autores,
                 cod_ISBN=None, data=None, num_exemplares=0,
                 suporte='Capa Dura'):

        # Validações
        if cod_ISBN and not self.cod_ISBN_valido(cod_ISBN):
            raise ValueError('ISBN %s inválido' % cod_ISBN)

        … etc …
```

18. Vamos agora acrescentar as funções/métodos para validar os outros itens em cima indicados. Após a definição de `cod_ISBN_valido`, acrescente o seguinte código:

```
    @staticmethod
    def titulo_valido(titulo):
        return bool(titulo)


    @staticmethod
    def genero_valido(genero):
        if not genero:
            return False
        return genero in {
            'Arte',
```

```
            'Biologia',
            'Ciências',
            'Economia',
            'Informática',
            'Literatura',
            'Matemática',
            'Tipografia',
        }

    @staticmethod
    def lista_autores_valida(autores):
        if not autores:
            return False
        return [autor for autor in autores if autor]

    @staticmethod
    def suporte_valido(suporte):
        if not suporte:
            return False
        return suporte in {
            'Capa Dura',
            'Capa Mole',
            'eBook',
        }
```

19. À semelhança de `cod_ISBN_valido`, acrescente as chamadas a estas funções de validação no código do mesmo construtor. Por exemplo:

```
class Livro:
        def __init__(self, titulo, genero, autores,
                 cod_ISBN=None, data=None, num_exemplares=0,
                 suporte='Capa Dura'):

            # Validações
            if not Livro.titulo_valido(titulo):
                raise ValueError('Titulo %s inválido' % titulo)

           if cod_ISBN and not self.cod_ISBN_valido(cod_ISBN):
               raise ValueError('ISBN %s inválido' % cod_ISBN)

            … acrescentar restantes validações …
```

20. Teste agora estas rotinas de validação alterando alguns parâmetros dos objectos criados anteriormente de modo a provocar as excepções.

21. Queremos acrescentar um identificador interno único a cada `Livro`. Este atributo deve ser incrementado automaticamente. Comece por acrescentar os seguinte atributos:

```
class Livro:
  conta_livros = 1

  def __init__(...etc...):
      id = conta_livros
      conta_livros += 1
```

Um atributo definido ao "nível" da classe, fora de qualquer método, é como um método definido com ***@staticmethod****: actua como uma variável global dentro da classe, sendo partilhada por todos os objectos dessa classe. Trata-se de uma* ***variável de classe*** *e não de instância.*

22. Finalmente, adicione uma linha ao método `mostra` para exibir o novo atributo `id`.

23. Vamos admitir que pretendemos obter o ano do `Livro`. Uma solução possível, mas pouco robusta e complicada, passa por:

```
>>> liv4.data.year
```

24. Outra solução passa por definir um método para aceder ao atributo ano. Por exemplo (utilizando a convenção de Java que passa por prefixar a palavra `get` a cada um destes métodos):

```
class Livro:
    def get_ano(self):
        return self.data.year
```

25. Mas em Python idiomático temos uma solução melhor: propriedades. Remova o método anterior e acrescente:

```
class Livro:
    @property
    def ano(self):
        return self.data.year
```

26. Agora pode aceder ao atributo (ie, à propriedade) ano como se fosse um campo:

```
>>> liv4.ano
2012
```

*Um "getter" é um método acessor que permite aceder a um atributo que pode ter que ser "reconstituido" a partir de um ou mais atributos. O objectivo de um método acessor passar por esconder a representação interna do atributo. Por exemplo, para uma dada uma classe `Circulo`, dois atributos importantes são o raio e o diâmetro. Um obtém-se a partir do outro. Podemos definir dois acessores, um para cada atributo, mas internamente apenas armazenamos o raio (por exemplo). Mais à frente podemos, por questões de eficiência, por exemplo, passar a armazenar o diâmetro em vez do raio. Neste caso, apenas precisamos de modificar os acessores respectivos, mas, mais importante, o código cliente da nossa classe `Circulo` não precisa de ser alterado uma vez que este código apenas utiliza os acessores. Em Java devemos definir um método `getAtributo` e outro `setAtributo` para cada atributo. Em Python e C# utilizamos "****propriedades****". Uma propriedade é um método ao qual acedemos como se de um campo se tratasse.*
Um "setter" é um método modificador que faz a operação inversa de um "getter", ou seja, permite modificar o valor de um atributo cuja representação interna necessita de um ou mais campos.

EXERCÍCIOS DE REVISÃO

1. O que é uma classe? E um objecto?

2. **`self`** é uma palavra-reservada?

3. O que é um decorador?

4. Os seguintes programas ou fragmentos de programas apresentam alguns erros. Corrija-os:

`classe C:`
`  def __init_(a, b):`
`  self.a = a`
`  self.b = b`	

`class D:`
`  def self(self, x)`
`  return self.a + self.b + x`
`  def __init__(self, a, b=10):`
`  self.a = a`
 	
`x = D()`
`x.a = 2`	

`class E:`
`  def __init__(self, x):`
`  self.a = x / 2.5`
`  self.b = 20`
`  @staticmethod`
`  def fromStr(self, str):`
`  return self()`	

EXERCÍCIOS DE PROGRAMAÇÃO

5. Uma aplicação de gestão de formação representa informação sobre formandos utilizando dicionários e funções. O módulo `formando` (`.py`), fornecido juntamente com este guia de laboratório, possui o código Python para este efeito. Crie uma versão deste módulo utilizando classes.

6. Uma aplicação de gestão necessita de lidar com informação sobre os seus colaboradores. Defina a classe `Colaborador` que deve possuir as seguintes propriedades:

. `prim_nome`: primeiro nome

. `apelido`

. `salario_anual`: vencimento anual bruto (assuma que desconta 15% de IRS e 11% de TSU)

. `num_meses_venc`: quantos meses de vencimento (apenas dois valores são aceites: 12 ou 14)

. `nome_completo`: método para devolver o nome completo do colaborador

. `salario_anual_liq`: método para devolver salário anual líquido

. `salario_mensal_liq`: método para devolver o salário líquido mensal

Desenvolva um construtor com todos os parâmetros obrigatórios para inicializar os campos de cada

objecto. Elabore código para testar esta classe, criando alguns objectos do tipo `Colaborador`.

7. Acrescente algumas validações básicas aos parâmetros.

8. Uma dada empresa de telecomunicações utiliza uma aplicação em Python para gestão de clientes. Assim um `Cliente` possui um primeiro nome, apelidos intermédios e apelido. Possui também idade, morada, código postal, NIF (número de identificação fiscal), e código de cliente. O nome completo deverá ser determinado a partir dos nomes todos. Deverá desenvolver uma operação para apresentar todos os dados de um cliente.

9. A mesma empresa guarda informação sobre as contas do cliente. Assim, cada conta possui código de conta (código alfanumérico único), morada, código postal, data de criação e uma referência para o cliente a quem a conta pertence. O valor da última factura também é guardado em cada conta. A esse valor acresce IVA que deverá ser calculado (utilize 22%).

10. Pretende implementar um catálogo de livros. Para tal, desenvolva uma classe (eg, `Catalogo` ou `ListaLivros`) especializada em gerir uma lista de livros. Para armazenar os livros em memória utilize uma das colecções do Python . Adicione os seguinte métodos:
. adicionar um livro não repetido
. pesquisas por ID, ISBN, título e género
. remover um livro do catálogo

11. Desenvolva uma aplicação gráfica (GUI) para gerir este catálogo utilizando uma biblioteca apropriada.