CENTRO DE FORMAÇÃO
DE ALCÂNTARA

INTRODUÇÃO À PROGRAMAÇÃO

UFCD(s) 5118, 5119

GUIA DE LABORATÓRIO 4.2
POO AVANÇADA (ßeta)

OBJECTIVOS

- Introduzir/Aprofundar os seguintes conceitos: **Herança**, **Sobreposição**, **Polimorfismo**, **Superclasse**, **Subclasse**, **Extensão**, (**Princípio da) Substituição**

INSTRUÇÕES

Contas bancárias: exemplo sem herança

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

2. Crie o ficheiro de código `banco1.py` .

Como vimos no laboratório anterior, as classes são o mecanismo por excelência que o Python oferece para implementar um TDA (Tipos de Dados Abstracto). A implementação de um TDA possui três componentes:
** Implementação das operações do TDA através de métodos*
** Implementação da(s) estrutura(s) de dado(s) que permitem representar valores desse tipo de dados*
** Estabelecimento dos invariantes que os métodos devem respeitar quando manipulam as estruturas de dados.*

*No contexto de POO (Programação Orientada por Objectos), **invariantes** são condições que se devem verificar ao longo da "vida" dos objectos. Por exemplo, ao implementarmos um TDA semelhante a um `set` (conjunto de elementos sem duplicados) utilizando listas para o efeito, um invariante importante consiste na não existência de elementos repetidos na lista. Todas as operações são responsáveis por manter este invariante do TDA. Em particular, as operações de inserção de elementos no conjunto não devem aceitar elementos repetidos e, para tal, devem validar se um novo elemento a inserir já existe na lista antes de o adicionarem.*

3. Vamos supor que pretendemos implementar um sistema de informação para um banco. Em particular, pretendemos apenas representar as estruturas para lidar com contas bancárias: Eis os requisitos:

- Existem quatro tipos de contas: Ordem, Ordenado, Prazo e Poupança-Habitação
- Para já, apenas estamos interessados em três operações bancárias básicas: obter do saldo, levantar e depositar
- Todas as contas são associaciadas a um cliente através de um número de cliente. Têm, também, um número de conta (que é o *id* de uma conta), uma data de abertura, um saldo e uma duração (decorrida). Opcionalmente, podem receber um número de conta exterior que, no caso de não ser utilizado, é gerado por um mecanismo de auto-numeração.
- A conta a prazo possui taxa de juro, duração mínima e saldo mínimo. Se a duração mínima ainda não tiver sido atingida, o saldo não deverá contemplar a aplicação de juros.

- A conta poupança-habitação é um tipo de conta a prazo, variando apenas nos valores da duração e saldo mínimos
- Para evitar complexidades adicionais 1) os juros são simples e não-compostos; 2) a contabilização dos juros é feita sempre do início, independentemente do número de depósitos feitos (isto é incorrecto para contas com juros, mas obrigaria a guardar uma lista de depósitos para calcular os valores correctos dos juros).
- A conta ordenado é um tipo de conta à ordem onde é possível ter um saldo negativo desde que não seja superior ao valor negativo do ordenado (ou seja, saldo >= -ordenado).
- Cada conta possui um estado com quatro valores: aberta, encerrada, bloqueada e inactiva.
- O método `__str__` deverá devolver uma string com os valores de todos os atributos em formato CSV; o método `__repr__` deverá devolver uma string com uma representação programática do respectivo objecto

Vamos começar por desenvolver uma solução sem utilizar herança e polimorfismo (os mecanismos da POO que pretendemos estudar neste laboratório).

Comece por adicionar as seguintes linha de código ao seu exemplo:

```
from decimal import Decimal
from datetime import date
from enum import Enum

TipoConta = Enum('TipoConta', 'ORDEM ORDENADO PRAZO POUPANCA_HABITACAO')

EstadoConta = Enum('EstadoConta', 'ABERTA ENCERRADA BLOQUEADA INACTIVA')

DATE_FMT = '%Y-%m-%d'
```

Enum *é um "mecanismo" existente a partir do Python 3.4 para criar enumerações de valores (não confundir com a função* ***enumerate****, que implementa um conceito diferente). Uma enumeração é um conjunto de valores simbólicos (ie, valores com um nome) constantes. Por exemplo, podemos definir uma enumeração para representar os dias da semana com:*

```
>>> DiaSemana = Enum('DiasSemana', 'SEGUNDA TERÇA QUARTA QUINTA SEXTA')
>>> dia = DiaSemana.SEGUNDA
>>> print(dia, dia.name, dia.value)
DiasSemana.SEGUNDA SEGUNDA 1
```

Cada valor da enumeração possui um valor inteiro associado (que podemos definir qual é, ainda que os valores sejam automaticamente atribuídos a partir de 1.). Por exemplo, neste caso o valor associado a SEGUNDA é 1. Alternativamente, podemos definir uma enumeração com uma classe que herda da classe ***Enum*** *(sim,* ***Enum*** *é um tipo de classe). Em outras versões de Python podemos obter o mesmo efeito com constantes numéricos ou textuais (ie, do tipo* ***str****) para os valores da enumeração:*

```
SEGUNDA = 1, TERCA = 2, etc. ou SEGUNDA = 'SEGUNDA', TERCA='TERÇA', etc.
```

Consultar: https://docs.python.org/3/library/enum.html

4. Vamos adicionar a classe `ContaBancaria` e a assinatura do método `__init__`:

```
class ContaBancaria:
```

```
    def __init__(
        self,
        num_cliente,
        saldo,
        num_conta=None,
        data_abertura=None,
        estado=EstadoConta.ABERTA,
        tipo=TipoConta.ORDEM,
        ordenado=Decimal('0.00'),
        taxa_juro=Decimal('0.00'),
    ):
        … continua já a seguir …
```

*O atributo `tipo` permite distinguir entre os tipos de contas. Uma vez que lidamos com valores monetários, utilizamos o tipo de dados **`Decimal`**. As taxas de juro são armazenadas em percentagem.*

5. Queremos introduzir algumas validações para os atributos em comum. Introduza:

```
class ContaBancaria:

    def __init__ (...):

        # Validações comuns
        if saldo < 0:
            raise ValueError("Saldo inicial %.2f inválido!" % saldo)

        if data_abertura and data_abertura < ContaBancaria.DATA_INICIAL:
            raise ValueError("Data %s inválida!" % data_abertura)

        if estado not in EstadoConta:
            raise ValueError("Estado inicial da conta %s inválido!" % estado)

        if tipo not in TipoConta:
            raise ValueError("Tipo de conta %s inválido!" % tipo)
```

6. Ao que se seguem algumas validações consoante o tipo de conta

*Como só temos uma classe um construtor **`__init__`**, este tem que contemplar todos os casos possíveis, o que torna a lógica do método muito complicada.*

```
class ContaBancaria:

    def __init__ (...):
        # ...

        # Validações conta à ORDEM e ORDENADO
        if tipo in (TipoConta.ORDENADO,) and ordenado < 0:
            raise ValueError("Ordenado %.2f inválido!" % ordenado)

        if tipo not in (TipoConta.ORDENADO,) and ordenado != 0:
            raise ValueError("Valor de ordenado %.2f inválido para este "
                             "tipo de conta %s!" % (ordenado, tipo))

        # Validações conta a PRAZO e POUPANCA_HABITACAO
        conta_com_juros = tipo in (TipoConta.PRAZO, TipoConta.POUPANCA_HABITACAO)
        poup_hab = tipo is TipoConta.POUPANCA_HABITACAO
```

```
        prazo = tipo is TipoConta.PRAZO

        if taxa_juro <= 0 and conta_com_juros:
            raise ValueError("Valor de juro %.2f inválido para este "
                             "tipo de conta %s!" % (taxa_juro, tipo))

        montante_min = ((poup_hab and ContaBancaria.saldo_min_phab) or
                        (prazo and ContaBancaria.saldo_min_prazo))
        if conta_com_juros and saldo < montante_min:
            raise ValueError("Saldo %.2f inválido para este "
                             "tipo de conta %s!" % (saldo, tipo))
```

7. Finalmente, vamos definir os atributos a partir dos parâmetros do **`__init__`**:

Guardamos o saldo num atributo "privado" `self._saldo`? Porquê? Porque o saldo é um valor que tem que ser calculado no caso das contas a prazo. Nestas contas, o atributo `self._saldo` armazena o saldo original sem aplicação de juros. Nas restantes contas, `self._saldo` armazena o saldo "de facto",
*À frente definimos uma **`property`** que devolve o valor correcto do saldo em função do tipo de conta. A duração também será uma **`property`**.*

```
class ContaBancaria:

    def __init__ (...):
        # ...

        self.num_cliente = num_cliente
        self._saldo = saldo
        if num_conta:
            self.num_conta = num_conta
        else:
            self.num_conta = ContaBancaria.prox_num_conta
            ContaBancaria.prox_num_conta += 1
        self.data_abertura = data_abertura if data_abertura else date.today()
        self.estado = estado
        self.tipo = tipo
        self.ordenado = ordenado
        self.taxa_juro = taxa_juro/100
```

A utilização de propriedades apenas para obtenção dos valores dos atributos, e não para os modificar, possibilita a definição de atributos de "leitura-apenas" (read only).

8. E agora as propriedades:

```
class ContaBancaria:

    def __init__ (...):
        # ...

    @property
    def saldo(self):
        assert self.tipo in TipoConta
        if self.tipo in (TipoConta.ORDEM, TipoConta.ORDENADO):
            return self._saldo
        else:
            if self.tipo is TipoConta.PRAZO:
                duracao_min = ContaBancaria.duracao_min_prazo
            else:
                duracao_min = ContaBancaria.duracao_min_phab
```

```
            dias = Decimal(self.duracao)
            saldo_com_juros = self._saldo * (1 + ((dias/365)*self.taxa_juro))
            return self._saldo if self.duracao < duracao_min else saldo_com_juros

    @property
    def duracao(self):
        return (date.today() - self.data_abertura).days
```

9. O método `levantar` também necessita de prestar atenção ao tipo de conta:

Note que levantar dinheiro de uma conta a prazo é diferente. Parte do que se levanta já "sofreu" o efeito dos juros. Temos que ter isso em atenção ao actualizamos o valor de `self._saldo` que, recorde-se, mantém o valor "original" do saldo sem juros.

```
class ContaBancaria:

    # __init__
    # saldo
    # duracao

    def levantar(self, montante):
        if montante < 0:
            raise ValueError("Montante %.2f inválido!" % montante)

        if self.tipo in (TipoConta.ORDEM, TipoConta.ORDENADO):
            if self.tipo is TipoConta.ORDENADO:
                saldo_min = -self.ordenado
            else:
                saldo_min = 0

            novo_saldo = self.saldo - montante
            if novo_saldo < saldo_min:
                raise ValueError("Saldo final %.2f inferior ao saldo mínimo %.2f!"
                                 % (novo_saldo, saldo_min))

        elif self.tipo in (TipoConta.PRAZO, TipoConta.POUPANCA_HABITACAO):
            if self.tipo is TipoConta.PRAZO:
                duracao_min = ContaBancaria.duracao_min_prazo
                saldo_min = ContaBancaria.saldo_min_prazo
            else:
                duracao_min = ContaBancaria.duracao_min_phab
                saldo_min = ContaBancaria.saldo_min_phab

            if self.duracao < duracao_min:
                raise ValueError("Prazo mínimo %s ainda não atingido!"
                                 % duracao_min)

            novo_saldo = self.saldo - montante
            if novo_saldo < saldo_min:
                raise ValueError("Saldo final %.2f inferior ao saldo mínimo %.2f!"
                                 % (novo_saldo, saldo_min))

            montante_sem_juros = montante / (1 + self.taxa_juro)
```

```
            novo_saldo_sem_juros = self._saldo - montante_sem_juros
            self._saldo = novo_saldo_sem_juros

        return self.saldo
```

10. E agora a nossa versão simplificada e incorrecta, no caso das contas a prazo, da operação `depositar`:

```
    def depositar(self, montante):
        # Versão incorrecta para contas a prazo (e derivadas) para
        # evitar tornar este código demonstrativo mais complexo
        if montante < 0:
            raise ValueError("Montante %.2f inválido!" % montante)
        self._saldo += montante
        return self.saldo
```

Por simplicidade, o método depositar não distingue entre os diversos tipos de conta.

11. Seguem-se as funções para representação dos objectos:

```
class ContaBancaria:

    # __init__
    # saldo
    # duracao
    # levantar e depositar

    def __str__(self):
        return ','.join((
                str(self.num_conta),
                str(self.num_cliente),
                self.tipo.name,
                "%.2f" % self.saldo,
                self.data_abertura.strftime(DATE_FMT),
                self.estado.name,
                "%.2f" % self.ordenado,
                "%.2f" % self.taxa_juro*100,
            ))

    def __repr__(self):
        return '\n'.join((
                "ContaBancaria(",
                "    tipo=%s," % self.tipo,
                "    num_conta=%r," % self.num_conta,
                "    num_cliente=%r," % self.num_cliente,
                "    saldo=%r," % self.saldo,
                "    data_abertura=%r," % self.data_abertura,
                "    estado=%s," % self.estado,
                "    ordenado=%r," % self.ordenado,
                "    taxa_juro=%r," % (self.taxa_juro*100),
```

```
                ")"
            ))
```

12. Finalmente, as variáveis de classe:

```
class ContaBancaria:

    # __init__
    # saldo
    # duracao
    # levantar e depositar
    # __str__ e __repr__

    DATA_INICIAL = date(1997, 5, 5)
    prox_num_conta = 117
    duracao_min_prazo = int(0.25*365)
    duracao_min_phab = 18*365
    saldo_min_prazo = 100
    saldo_min_phab = 150
```

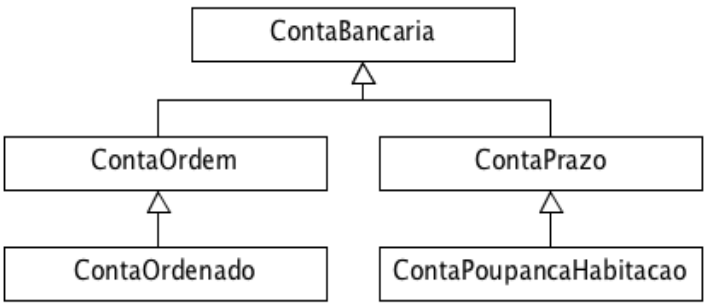
Esta implementação, apesar de completa (em termos dos requisitos iniciais), apresenta alguns problemas:
. a lógica é complexa de seguir devido à necessidade frequente de consultar o tipo de conta
. é difícil de alterar e manter, porque o código para os diferentes tipos de conta está misturado.
Vamos supor que era necessário adicionar um tipo de conta a prazo novo, com prazos e taxas diferentes e com penalização de juros. O que seria necessário alterar?

Contas bancárias: exemplo com herança

13. Vamos agora abordar este problema por outra perspectiva. De facto, nos requisitos mencionámos que:

- Todos os tipos de conta partilham atributos e algum comportamento entre si, mas depois diferem em "pequenos" aspectos.
- Na prática, temos dois tipos de contas bancárias: a prazo e à ordem. Os outros dois tipos de conta são derivações destes dois tipos.

O esquema gráfico seguinte relaciona os tipos de contas entre si:



*Este diagrama indica que uma `ContaOrdenado` relaciona-se com `ContaOrdem` através de uma relação que designamos por **herança** (**inheritance**) ou **derivação** (**subclassing**). De acordo com esta relação, dizemos que `ContaOrdenado` herda ou deriva de `ContaOrdem`. Esta relação também pode ser descrita pela expressão "**é-uma**" ("**is-a**"): uma conta à ordenado **é uma** conta à ordem. Note-se que a relação entre `ContaOrdem` e `ContaBancaria` é idêntica. Logo, também podemos afirmar que `ContaOrdenado` herda (ou deriva) de `ContaBancaria` e, como tal, é uma `ContaBancaria`. As relações são semelhantes no lado das contas a prazo.*
*No jargão da POO, a classe que herda (ou deriva) é designada de **subclasse** , a classe herdada (ou derivada) recebe o nome de **superclasse**.*

*Dois dos pilares da Programação Orientada por Objectos são **encapsulamento** e **ocultação de informação**.*
***Herança** é também um dos conceitos fundamentais da POO. Herança possibilita o seguinte:*
. um mecanismo hierárquico para relacionar classes (ie, abstracções)
. uma forma de reutilização de código, uma vez que uma classe que herda de outra ganha acesso aos seus atributos e métodos (desde que estes sejam "herdáveis")
. um mecanismo de especialização: uma subclasse pode especializar um determinado comportamento da superclasse (eg, a operação de levantamento das contas ordenado é uma especialização da operação de uma conta ordem).
*. uma possibilidade de substituir um objecto de uma superclasse por outro de uma subclasse, sem que código cliente tenha que ser alterado (**princípio da Substiuição de Liskov**).*
Para além dos atributos e métodos herdados, uma subclasse pode:
*. **adicionar** novos métodos e atributos*
*. **redefinir** ou **sobrepor** (**overriding**) novos métodos e atributos com intuito de os especializar para determinada finalidade*
*. **extender** métodos existentes, acrescentando-lhes novas funcionalidades*
*Uma outra noção importante em POO é a noção de **polimorfismo** (múltiplas formas, em inglês: **polymorphism**). Basicamente, consiste na possibilidade de uma "operação" se adaptar a argumentos de tipos de dados diferentes. Em Python, já lidámos com várias funções polimórficas: **sum**, **max**, **enumerate**, etc.(todas se adaptam a diferentes tipos de colecções de elementos). A POO acrescenta uma outra dimensão ao polimorfismo: a possibilidade de definirmos métodos numa subclasse com o mesmo nome que métodos na superclasse; o código cliente consegue chamar a implementação correcta através do tipo do objecto à esquerda do ponto.*

14. Crie o ficheiro `banco2.py`.

15. Acrescente o seguinte código inicial:

```
from decimal import Decimal
from datetime import date
from enum import Enum

EstadoConta = Enum('EstadoConta', 'ABERTA ENCERRADA BLOQUEADA INACTIVA')

DATE_FMT = '%Y-%m-%d'
```

16. De seguida, acrescente a superclasse na raiz da nossa hierarquia de classes e o respectivo construtor:

Esta classe apenas tem o o que é comun a todas as contas bancárias.

```
class ContaBancaria:

    def __init__(
        self,
        num_cliente,
        saldo,
        num_conta=None,
        data_abertura=None,
        estado=EstadoConta.ABERTA,
    ):
        if saldo < self.saldo_min:
            raise ValueError("Saldo inicial %.2f inválido!" % saldo)
```

```
        if data_abertura and data_abertura < ContaBancaria.DATA_INICIAL:
            raise ValueError("Data %s inválida!" % data_abertura)

        if estado not in EstadoConta:
            raise ValueError("Estado inicial da conta %s inválido!" % estado)

        self.num_cliente = num_cliente
        self._saldo = saldo
        if num_conta:
            self.num_conta = num_conta
        else:
            self.num_conta = ContaBancaria.prox_num_conta
            ContaBancaria.prox_num_conta += 1
        self.data_abertura = data_abertura if data_abertura else date.today()
        self.estado
```

17. Em seguida seguem-se as propriedades:

```
class ContaBancaria:

    # __init__

    @property
    def duracao(self):
        return (date.today() - self.data_abertura).days

    def depositar(self, montante):
        # Versão incorrecta para contas a prazo (e derivadas) para
        # evitar tornar este código demonstrativo mais complexo
        if montante < 0:
            raise ValueError("Montante %.2f inválido!" % montante)
        self._saldo += montante
        return self.saldo
```

18. E agora as funções de **__str__** e **__repr__** preparadas para serem extendidas via herança:

```
class ContaBancaria:

    # __init__
    # @propriedades e depositar

    def __str__(self):
        return ','.join((
                str(self.num_conta),
                str(self.num_cliente),
                self.__class__.__name__,
                "%.2f" % self.saldo,

self.data_abertura.strftime(DATE_FMT),
```

*O atributo **__class__**, que qualquer objecto possui, indica qual a sua classe.*
*Uma classe é um objecto do tipo **type**, e estes objectos possuem o atributo __name__ que lhes indica o nome da classe. Ou seja, **self.__class__.__name__** indica qual o nome da classe do objecto **self**. Como veremos, esta função adapta-se a classes que derivem desta uma vez que o objecto **self.__class__** vai ser diferente. A função _attrsReprs foi criada reunir todos os atributos legendados de uma classe. Vamos extender esta função nas classes derivadas sempre que estas acrescentarem novos atributos.*

```
                self.estado.name,
            ))

    def __repr__(self):
        txt = '\n  '.join(self._attrsReprs())
        return "%s(\n  %s\n)" % (self.__class__.__name__, txt)

    def _attrsReprs(self):
        return [
            "num_conta=%r," % self.num_conta,
            "num_cliente=%r," % self.num_cliente,
            "saldo=%r," % self.saldo,
            "data_abertura=%r," % self.data_abertura,
            "estado=%s," % self.estado,
        ]
```

19. As variáveis de classe da `ContaBancaria`:

```
class ContaBancaria:

    # __init__
    # @propriedades e depositar
    # __str__ e __repr__

    DATA_INICIAL = date(1997, 5, 5)
    prox_num_conta = 117
    saldo_min = 0
```

20. A primeira subclasse será `ContaOrdem`:

Indicamos que classe B herda de A fazendo:

```
class B(A):
 #...
```

*Neste caso, `ContaOrdem` herda o **`__init__`**, a `duracao`, o `depositar`, as funções de representação textual e as variáveis de classe. Vamos acrescentar a operação `levantar`.*

```
class ContaOrdem(ContaBancaria):

    @property
    def saldo(self):
        return self._saldo

    def levantar(self, montante):
        if montante < 0:
            raise ValueError("Montante %.2f inválido!" % montante)
        if self._saldo < montante:
            raise ValueError("Saldo insuficiente %.2f" % self._saldo)
        self._saldo -= montante
```

21. Agora vem `ContaOrdenado` que é mais complexa porque temos um novo atributo, `ordenado`, e porque temos que lidar com o saldo negativo (desde que superior a `-ordenado`).

```
class ContaOrdenado(ContaOrdem):
```

```
    def __init__(self, ordenado, *args_pos, **args_com_nome):
        if ordenado <= 0:
            raise ValueError("Ordenado %.2f inválido!" % ordenado)
        super().__init__(*args_pos, **args_com_nome)
        self.ordenado = ordenado

    def levantar(self, montante):
        if montante < 0:
            raise ValueError("Montante %.2f inválido!" % montante)
        novo_saldo = self._saldo - montante
        if novo_saldo < -self.ordenado:
            raise ValueError("Saldo insuficiente %.2f" % self._saldo)
        self._saldo -= montante

    def __str__(self):
        txt = super().__str__()
        return txt + (",%.2f" % self.ordenado)

    def _attrsReprs(self):
        fields = super()._attrsReprs()
        fields.append("ordenado=%s," % self.ordenado)
        return fields
```

*A função built-in **super** permite aceder a um objecto especial que, por seu turno, permite aceder a um método de uma superclasse (a partir da classe mais baixa na hierarquia de herança). Este método, definido numa outra superclasse, será invocado sobre o objecto que está a ser manipulado quando o super é invocado. Isto é útil para aceder métodos de uma superclasse que foram redefinidos pela subclasse.*
Consultar a documentação oficial do Python (2. Built-in Functions) e localizar o documento "Python's Super Considered Super".

22. Sem mais demoras, acrescentamos o código das últimas duas classes: `ContaPrazo` e `ContaPoupancaHabitacao`:

```
class ContaPrazo(ContaBancaria):

    def __init__(self, taxa_juro, *args_pos, **args_com_nome):
        if taxa_juro < 0:
            raise ValueError("Taxa de juro %.2f inválida!" % taxa_juro)
        super().__init__(*args_pos, **args_com_nome)
        self.taxa_juro = taxa_juro/100

    @property
    def saldo(self):
        if self.duracao < self.duracao_min:
            return self._saldo
        dias = Decimal(self.duracao)
        return self._saldo * (1 + ((dias/365)*self.taxa_juro))

    def levantar(self, montante):
        if montante < 0:
            raise ValueError("Montante %.2f inválido!" % montante)

        if self.duracao < self.duracao_min:
            raise ValueError("Prazo mínimo %s dias ainda não atingido!"
                             % self.duracao_min)
```

```
            novo_saldo_com_juros = self.saldo - montante
            if novo_saldo_com_juros < self.saldo_min:
                raise ValueError("Saldo final %.2f inferior ao saldo mínimo %.2f!"
                                 % (novo_saldo_com_juros, self.saldo_min))

            montante_sem_juros = montante / (1+self.taxa_juro)
            novo_saldo_sem_juros = self._saldo - montante_sem_juros
            self._saldo = novo_saldo_sem_juros
            return self.saldo

        def __str__(self):
            txt = super().__str__()
            return txt + (",%.2f" % (self.taxa_juro*100))

        def _attrsReprs(self):
            fields = super()._attrsReprs()
            fields.append("taxa_juro=%r," % (self.taxa_juro*100))
            return fields

        duracao_min = int(0.25*365)
        saldo_min = 100


    class ContaPoupancaHabitacao(ContaPrazo):
        duracao_min = 18*365
        saldo_min = 150
```

23. Teste o código convenientemente.

EXERCÍCIOS DE REVISÃO

1. Considere o seguinte bloco de código: (**NOTA**: Também distribuído à parte)

```
class Pessoa:

    def __init__(self, nome):
        self.nome = nome

    def apresenteSe(self):
        return "Eu sou o/a " + self.nome + "."


class PessoaFormal(Pessoa):

    def __init__(self, nome, titulo):
        # - 1 -
        self.titulo = titulo

    def apresenteSe(self):
        return super().apresenteSe() + " Ao seu dispor."

    def obtemTitulo(self):
        return self.titulo
```

```
def teste():
    p = Pessoa("Alberto")
    print(p.apresenteSe())

    p = PessoaFormal("Armando", "Doutor")
    print(p.apresenteSe())
```

1.1 O que é exibido pela função `teste`? Terá que substituir a(s) zona(s) assinalada(s) com número(s) pelas instruções correctas.

2. Considere o seguinte bloco de código: (**NOTA**: Também distribuído à parte)

```
class Colaborador:

    def __init__(self, sal_base):
        self.sal_base = sal_base

    def vencimento(self):
        return self.sal_base + self.obtemBonus()

    def obtemBonus(self):
        return 50


class ColaboradorSenior(Colaborador):

    def __init__(self, sal_base):
        super().__init__(sal_base)


    def obtemBonus(self):
        return 200
```

```
def teste1():
    c = Colaborador(Decimal('1000'))
    print(c.vencimento())

    c = ColaboradorSenior(Decimal('1000'))
    print(c.vencimento())
```

2.1 O que é exibido pela função `teste1`?

2.2 É suposto o `ColaboradorSenior` receber o bónus dado aos `Colaborador`es (ou seja, ele deveria ter um bónus de 250). Neste sentido, quais as alterações (mínimas) a introduzir no programa?

2.3 O construtor em `ColaboradorSenior` é mesmo necessário? Porquê?