Universidade Federal de Roriama
Departamento de Ciência da Computação
Professor: Filipe Dwan Pereira
Código da disciplina: DCC305
Período: 2019.2

Disclaimer

Esta aula é uma adaptação do capítulo 5 do livro:

- Phillips, Dusty. Python 3 Object-oriented Programming - Unleash the power of Python 3 objects. "Packt Publishing", 2015. Second Edition.

- Nesta aula aprenderemos:
  - Juntando dados e comportamento usando *properties*.
  - Restringir dados usando comportamento;
  - Não permitir código repetido;

Adicionando comportamento as dados da classe com properties

- Em linguagens de programação orientada a objetos é comum que atributos sejam acessados via métodos.
- Para tanto, tais linguagens usam modificadores de visibilidade, deixando os atributos como privados e os métodos que dão acesso aos atributos como públicos.
- Por exemplo, a classe Conta simula esse conceito:

In [1]:

```
class Conta:
    def __init__(self, titular):
        self._titular = titular;
        self._saldo = 0.0
        self._limite = 100.0
    def set_titular(self, titular):
        if not isinstance(titular, str):
            raise TypeError('Titular:: entre com uma string')
        if len(titular) < 3:
            raise TypeError('Nome:: entre com seu nome completo')
        self._titular = titular;
    def get_titular(self):
        return self._titular
    def set_limite(self, limite):
        if limite<0:
            raise ValueError('Limite::Valor Inválido: ', limite)
        self._limite = limite;
    def get_limite(self):
        return self._limite
    def get_saldo(self):
        return self._saldo + self._limite
    '''
    set_saldo() não faz sentido, pois devemos usar os métodos
    depositar e sacar para modificar o saldo:
    '''
    def depositar(self, valor):
        if valor<0:
            raise ValueError('Depósito::Valor Inválido: ', valor)
        self._saldo+=valor
    def sacar(self, valor):
        if valor<0:
            raise ValueError('Sacar::Valor Inválido: ', valor)
        if valor > (self._saldo + self._limite):
            raise ValueError('Sacar::Saldo Insuficiente')
        self._saldo-=valor
```

- Primeiramente, vamos testar nossa classe:

In [2]:

```
conta1 = Conta('Filipe')
conta1.depositar(500)
conta1.sacar(200)
conta1.get_saldo()
```

Out[2]:

```
400.0
```

- O underscore prefixado nos atributos sugere que eles sejam "privados";
- Além disso, os métodos gettters and setters são públicos e dão acesso a tais atributos "privados";
- Usar getters e setters pode parecer trocar 6 por meia dúzia, mas essas linguagens fazem isso por um motivo:
  - Por exemplo, um banco mal intencionado (como na classe que fizemos) pode retornar o saldo + limite no método *get_saldo*, para incentivar o cliente a entrar no cheque especial;
  - Em outras palavras, não necessariamente o método getX() vai retornar X;

    - Além disso, observe os métodos *set_titular* e *set_limite*, onde validamos os valores de entrada antes de simplemente atribuí-los aos atributos;
- Por fim, perceba que não faz sentido termos um método *set_saldo*, pois movimentaramos o saldo usando os métodos depositar e sacar;

- Em python, como já sabemos, não existe conceito de membros privados;
    - Assim podemos quebrar o código da classe Conta, acessando diretamente os atributos.
- Veja:

In [30]:

```
conta2 = Conta(titular='Fulano')
conta2._saldo = -1000 #deveria ser um valor inválido
conta2._saldo #deveria mostrar o saldo+limite
```

Out[30]:

```
-1000
```

- O python permite a implementação desses conceitos importantíssimos de linguagens orientada a objetos.
    - No python, fazer isso sem precisar mudar a interface pública do usuário, usando a palavra-chave **property**;
- Em outras palavras, em python o usuário acessa ao atributo X, com um simples **.x**, ao invés de **.get_X()**;
- Para tanto, o python realiza uma fusão de dados e comportamento usando property:
    - Os atributos poderão ter comportamentos também.
    - Para tanto, devemos associar o atributo X com os métodos get_X() e set_X();
- Veja o nosso exemplo da classe Conta usando property:

In [4]:

```
class Conta:
    def __init__(self, titular):
        self._titular = titular;
        self._saldo = 0.0
        self._limite = 100.0
    def _set_titular(self, titular):
        if not isinstance(titular, str):
            raise TypeError('Titular:: entre com uma string')
        if len(titular) < 3:
            raise TypeError('Nome:: entre com seu nome completo')
        self._titular = titular;
    def _get_titular(self):
        return self._titular
    def _set_limite(self, limite):
        if limite<0:
            raise ValueError('Limite::Valor Inválido: '+ str(limite))
        self._limite = limite;
    def _get_limite(self):
        return self._limite
    def _get_saldo(self):
        return self._saldo + self._limite
    def _set_saldo(self, saldo):
        raise Exception('Saldo:: use os métodos Sacar e Depositar para movime
    def depositar(self, valor):
        if valor<0:
            raise ValueError('Depósito::Valor Inválido: '+ str(valor))
        self._saldo+=valor
    def sacar(self, valor):
        if valor<0:
            raise ValueError('Sacar::Valor Inválido: '+ str(valor))
        if valor > (self._saldo + self._limite):
            raise ValueError('Sacar::Saldo Insuficiente')
        self._saldo-=valor

    titular = property(_get_titular, _set_titular)
    limite = property(_get_limite, _set_limite)
    saldo = property(_get_saldo, _set_saldo)
```

- Nas linhas [35-37] foram criadas as propriedades titular, limite e saldo.
  - Agora ao acessar *objeto_conta.titular*, você na verdade estará acessando indiretamente o método *_get_titular*;
  - Ao atribuir um valor ao *objeto_conta.titular*, você estará usando inderamente o método *set_titular*;
- Essas propriedades funcionarão como atributos que possuem regras:
  - Por exemplo, Não podemos mais atribuir valores negativos para o saldo:

In [5]:

```
conta2 = Conta(titular='Fulano')
conta2.saldo = -1000
```

```
---------------------------------------------------------------------
-----
Exception                                 Traceback (most recent call
 last)
<ipython-input-5-caa60bad367c> in <module>
      1 conta2 = Conta(titular='Fulano')
----> 2 conta2.saldo = -1000

<ipython-input-4-7dff6f6bea15> in _set_saldo(self, saldo)
     21         return self._saldo + self._limite
     22     def _set_saldo(self, saldo):
---> 23         raise Exception('Saldo:: use os métodos Sacar e Deposi
tar para movimentar o saldo')
     24     def depositar(self, valor):
     25         if valor<0:

Exception: Saldo:: use os métodos Sacar e Depositar para movimentar o
 saldo
```

- Não podemos atribuir nomes muito curtos (menos que 3 letras) para o titular:

In [6]:

```
conta2.titular = 'a'
```

```
---------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call
 last)
<ipython-input-6-c155891a8f02> in <module>
----> 1 conta2.titular = 'a'

<ipython-input-4-7dff6f6bea15> in _set_titular(self, titular)
      8             raise TypeError('Titular:: entre com uma string')
      9         if len(titular) < 3:
---> 10             raise TypeError('Nome:: entre com seu nome complet
o')
     11         self._titular = titular;
     12     def _get_titular(self):

TypeError: Nome:: entre com seu nome completo
```

- Também não podemos atribuir valores negativos pra o limite da conta:

In [19]:

```
1  conta2.limite = -100
```

```
---------------------------------------------------------------------
-----
ValueError                                Traceback (most recent call
 last)
<ipython-input-19-beb87490df24> in <module>
----> 1 conta2.limite = -100

<ipython-input-16-7dff6f6bea15> in _set_limite(self, limite)
     14     def _set_limite(self, limite):
     15         if limite<0:
---> 16             raise ValueError('Limite::Valor Inválido: '+ str(l
imite))
     17         self._limite = limite;
     18     def _get_limite(self):

ValueError: Limite::Valor Inválido: -100
```

- Lembre-se que mesmo com uso de property, o código ainda não está totalmente seguro;
- Ainda é possível acessar o _nameattribute diretamente.
- Mas se alguém acessar um atributo marcado com undescore para sugerir que é privado, esse alguém que deve arcar com as consequências disso, não quem fez a classe.

In [20]:

```
1  conta2._limite = -100 ##não faça isso
```

In [21]:

```
1  conta2.limite
```

Out[21]:

```
-100
```

Um pouco mais sobre property

- Além dos setters e getters, O property pode aceitar mais dois argumentos:
  1. Um método para remoção do atributo
  2. O docstring da propriedade
- Nota: a função delete é raramente utilizada;
- Veja o exemplo abaixo:

In [9]:

```
class Cliente:
    def _get_nome(self):
        print('Você acessou o nome do cliente')
        return self._nome
    def _set_nome(self, nome):
        print('O nome do cliente eh: '+nome)
        self._nome = nome
    def _del_nome(self):
        print('Você deletou o nome do cliente')
        del self._nome
    nome = property(_get_nome, _set_nome,
                    _del_nome, "docstring: nome do cliente")
```

In [10]:

```
cliente = Cliente()
cliente.nome = 'Ciclano'
```

```
O nome do cliente eh: Ciclano
```

In [11]:

```
cliente.nome
```

```
Você acessou o nome do cliente
```

Out[11]:

```
'Ciclano'
```

In [12]:

```
del cliente.nome
```

```
Você deletou o nome do cliente
```

In [13]:

```
cliente.nome
```

```
Você acessou o nome do cliente

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call
 last)
<ipython-input-13-af3724e1243a> in <module>
----> 1 cliente.nome

<ipython-input-9-45a6e15e0e9b> in _get_nome(self)
      2     def _get_nome(self):
      3         print('Você acessou o nome do cliente')
----> 4         return self._nome
      5     def _set_nome(self, nome):
      6         print('O nome do cliente eh: '+nome)

AttributeError: 'Cliente' object has no attribute '_nome'
```

Properties com Decorators

- Outra forma de criar properties é usando decorators.
  - Com decorators, o código fica mais fácil de ler;
- Veja abaixo um exemplo para que tornar um getter em uma property:

In [14]:

```
class Cliente:
    def __init__(self, nome):
        self._nome = nome
    @property
    def nome(self):
        print('Acessando o nome do cliente')
        return self._nome
```

In [15]:

```
cliente = Cliente('Fulano de Tal')
```

In [16]:

```
cliente.nome
```

```
Acessando o nome do cliente
```

Out[16]:

```
'Fulano de Tal'
```

- Ao usarmos @property, nós não precisamos fazer:
  - nome = property(_get_nome)
- Vamos refazer as propriedades da nossa conta usando decorators:

In [47]:

```
class Conta:
    def __init__(self, titular):
        "informe o nome do titular da conta"
        self._titular = titular;
        self._saldo = 0.0
        self._limite = 100.0
    @property
    def titular(self):
        "nome do titular da conta"
        return self._titular
    @titular.setter
    def titular(self, titular):
        if not isinstance(titular, str):
            raise TypeError('Titular:: entre com uma string')
        if len(titular) < 3:
            raise TypeError('Nome:: entre com seu nome completo')
        self._titular = titular;
    @property
    def limite(self):
        "limite da conta para cheque especial"
        return self._limite
    @limite.setter
    def limite(self, limite):
        if limite<0:
            raise ValueError('Limite::Valor Inválido: '+ str(limite))
        self._limite = limite;
    @property
    def saldo(self):
        return self._saldo + self._limite
    @saldo.setter
    def saldo(self, saldo):
        raise Exception('Saldo:: use os métodos Sacar e Depositar para movime
    def depositar(self, valor):
        if valor<0:
            raise ValueError('Depósito::Valor Inválido: '+ str(valor))
        self._saldo+=valor
    def sacar(self, valor):
        if valor<0:
            raise ValueError('Sacar::Valor Inválido: '+ str(valor))
        if valor > (self._saldo + self._limite):
            raise ValueError('Sacar::Saldo Insuficiente')
        self._saldo-=valor
```

In [50]:

```
nova_conta = Conta('Beltrano')
```

In [52]:

```
nova_conta.limite = 1000
```

In [53]:

```
1  nova_conta.limite
```

Out[53]:

```
1000
```

In [54]:

```
1  nova_conta.saldo = 1000
```

```
---------------------------------------------------------------------
-----
Exception                                 Traceback (most recent call
 last)
<ipython-input-54-ca61ca47b353> in <module>
----> 1 nova_conta.saldo = 1000

<ipython-input-47-7449868a280f> in saldo(self, saldo)
     30     @saldo.setter
     31     def saldo(self, saldo):
---> 32         raise Exception('Saldo:: use os métodos Sacar e Deposi
tar para movimentar o saldo')
     33     def depositar(self, valor):
     34         if valor<0:

Exception: Saldo:: use os métodos Sacar e Depositar para movimentar o
 saldo
```

Caso de Uso

- Neste estudo de caso criaramos uma classe Document que representa um documento que pode ser processado por um editor de texto;
- Precisaremos de uma string para armazenar o conteúdo do documento;
  - No entanto, strings são imutáveis em python e, assim, é inviável deixarmos nosso atributo conteúdo como string;
  - Faremos dele uma lista de caracteres;
- Precisaremos ainda de um cursor que apontará para o caractere corrente na lista;
- Por fim, precisaremos armazenar o nome do documento;
- Precisaremos de alguns métodos simples como para inserir um caractere, deletar e etc.
- Veja abaixo nossa primeira versão da classe Documento:

In [4]:

```
class Document:
    def __init__(self):
        self.characters = []
        self.cursor = 0
        self.filename = ''
    def insert(self, character):
        self.characters.insert(self.cursor, character)
        self.cursor += 1
    def delete(self):
        del self.characters[self.cursor]
    def save(self):
        with open(self.filename, 'w') as f:
            f.write(''.join(self.characters))
    def forward(self):
        self.cursor += 1
    def back(self):
        self.cursor -= 1
```

- Vamos testar nossa classe:

In [5]:

```
doc = Document()
doc.filename = 'documento_teste.txt'
doc.insert('p')
doc.insert('r')
doc.insert('o')
doc.insert('g')
doc.insert('r')
doc.insert('a')
doc.insert('m')
doc.insert('a')
doc.insert('r')
doc.insert(' ')
doc.insert('é')
doc.insert(' ')
doc.insert('b')
doc.insert('o')
doc.insert('m')
doc.save()
```

In [6]:

```
!cat documento_teste.txt
```

```
programar é bom
```

In [7]:

```
doc.back()
doc.delete()
doc.insert('M')
doc.save()
```

In [8]:

```
!cat documento_teste.txt
```

```
programar é boM
```

- Observe que em um teclado existem, além do forward e back, as teclas end, home, page up e etc.
- Assim sendo, faz mais sentido o cursos ser uma classe que consegue se locomover pelo documento dependendo da opção usada:

In [10]:

```
class Cursor:
    def __init__(self, document):
        self.document = document
        self.position = 0
    def forward(self):
        self.position += 1
    def back(self):
        self.position -= 1
    def home(self):
        while self.document.characters[self.position-1] != '\n':
            self.position -= 1
            if self.position == 0:
                break
    def end(self):
        while self.position < len(self.document.characters) and self.document
            self.position += 1
```

- Precisamos atualizar nossa classe Document:
  - Vamos adicionar, além do objeto cursor, uma propriedade para mostrar o conteúdo do documento;

In [11]:

```
class Document:
    def __init__(self):
        self.characters = []
        self.cursor = Cursor(self)
        self.filename = ''
    def insert(self, character):
        self.characters.insert(self.cursor.position,character)
        self.cursor.forward()
    def delete(self):
        del self.characters[self.cursor.position]
    def save(self):
        f = open(self.filename, 'w')
        f.write(''.join(self.characters))
        f.close()
    @property
    def content(self):
        return "".join(self.characters)
```

In [12]:

```
doc = Document()
doc.filename = 'documento_teste.txt'
doc.insert('p')
doc.insert('r')
doc.insert('o')
doc.insert('g')
doc.insert('r')
doc.insert('a')
doc.insert('m')
doc.insert('a')
doc.insert('r')
doc.insert('\n')
doc.insert('é')
doc.insert('\n')
doc.insert('b')
doc.insert('o')
doc.insert('m')
print(doc.content)
```

```
programar
é
bom
```

In [13]:

```
doc.cursor.home()
doc.insert('+')
```

In [14]:

```
print(doc.content)
```

```
programar
é
+bom
```

- Vamos aumentar um pouco mais as funcionalidades do documento;
- Queremos ter caracteres em negrito, sublinhado e itálico;
- Para tanto, criaremos uma classe caractere que terá esses atributos;
  - Dessa forma, se o atributo negrito estiver configurado como True, significa que o caractere está em negrito;
- Além disso, iremos sobreescrever o método __str__ da classe object.
  - Assim, ao invés de imprimir o nome da classe e o endereço de memória do objeto, podemos imprimir uma mensagem significativa do objeto dentro de um comando print. Veja um simple exemplo:

In [15]:

```
class Pessoa:
    def __init__(self, nome):
        self.nome = nome
```

In [16]:

```
p = Pessoa('Fulano')
print(p)
```

```
<__main__.Pessoa object at 0x7f31fc80ddd8>
```

In [17]:

```
class Pessoa:
    def __init__(self, nome):
        self.nome = nome
    def __str__(self):
        return 'Meu nome é '+self.nome
```

In [38]:

```
p = Pessoa('Fulano')
print(p)
```

```
Meu nome eh Fulano
```

- Agora vamos voltar para a nossa classe que representa um caractere do documento:

In [19]:

```
class Character:
    def __init__(self, character,bold=False, italic=False, underline=False):
        assert len(character) == 1
        self.character = character
        self.bold = bold
        self.italic = italic
        self.underline = underline
    def __str__(self):
        #colocaremos um prefixo em caracteres formatados
        bold = "*" if self.bold else ''
        italic = "/" if self.italic else ''
        underline = "_" if self.underline else ''
        return bold + italic + underline + self.character
```

- Agora precisamos fazer pequenas mudanças no método insert e na propriedade content da classe documento;
- Além disso mudaremos os métodos home e end para acessarmos os atributos *character* de cada objeto da classe Character;
- Nosso módulo ficará assim:

In [20]:

```
class Document:
    def __init__(self):
        self.characters = []
        self.cursor = Cursor(self)
        self.filename = ''
    def insert(self, character):
        if not hasattr(character, 'character'):
            character = Character(character)
        self.characters.insert(self.cursor.position,character)
        self.cursor.forward()
    def delete(self):
        del self.characters[self.cursor.position]
    def save(self):
        f = open(self.filename, 'w')
        f.write(''.join(self.characters))
        f.close()
    @property
    def content(self):
        for c in self.characters:
            print(c, end='')
        return ''

class Character:
    def __init__(self, character,bold=False, italic=False, underline=False):
        assert len(character) == 1
        self.character = character
        self.bold = bold
        self.italic = italic
        self.underline = underline
    def __str__(self):
        #colocaremos um prefixo em caracteres formatados
        bold = "*" if self.bold else ''
        italic = "/" if self.italic else ''
        underline = "_" if self.underline else ''
        return bold + italic + underline + self.character

class Cursor:
    def __init__(self, document):
        self.document = document
        self.position = 0
    def forward(self):
        self.position += 1
    def back(self):
        self.position -= 1
    def home(self):
        while self.document.characters[self.position-1].character != '\n':
            self.position -= 1
            if self.position == 0:
                break
    def end(self):
        while self.position < len(self.document.characters) and self.document
            self.position += 1
```

- Agora vamos testar nosso código:

In [21]:

```
doc = Document()
doc.filename = 'documento_teste_2.txt'
doc.insert('p')
doc.insert('r')
doc.insert(Character('o', bold=True))
doc.insert(Character('g', italic=True))
doc.insert('r')
doc.insert('a')
doc.insert('m')
doc.insert(Character('a', underline=True))
doc.insert('r')
doc.insert('\n')
doc.insert('é')
doc.insert('\n')
doc.insert('b')
doc.insert('o')
doc.insert('m')
print(doc.content)
```

```
pr*o/gram_ar
é
bom
```

Nota: O *hasattr* verifica se o valor passado como parâmetro possui o atributo *character*;

- Caso não tenha, encapsulamos ele em um objeto Character;
- Para ilustrar, veja abaixo um exemplo de uso dessa função:

In [48]:

```
class Pessoa:
    nome = 'Fulano'
    idade = 28

p = Pessoa()
print(hasattr(p, 'nome'))#verifica se o objeto p tem o atributo nome -> True
print(hasattr(p, 'altura'))#verifica se o objeto p tem o atributo altura -> F
```

```
True
False
```