

## Disclaimer

Esta aula é uma adaptação do capítulo 3 do livro:

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

Nesta aula aprenderemos:

- O básico sobre herança
- Usar herança a partir de built-ins
- Herança Múltipla;
- Polimorfismos e duck typing;

## Herança Básica

- Tecnicamente, todas as classes que usamos já usam herança, uma vez que elas herdam da classe *object*;
  - Os comportamentos fornecidos por essa classe são aqueles com underscore duplo;
- A classe *object* permite que o python trate todos os objetos da mesma maneira, isto é, como um objeto;
- Até agora ao criar nossas classes não explicitamos que herdamos de *object*, mas podemos, conforme segue:

In [1]:

```
1 class MySubClass(object):  
2     pass
```

- Isso é **herança**;
  - Chamamos de superclasse ou classe pai a classe que estendemos e de subclasse ou classe filha a classe que herda.
    - No caso acima, *object* é a superclasse e *MySubClasse* a subclasse;
  - Para realizar a herança em python basta colocar o nome da superclasse a qual se quer herdar dentro dos parênteses da definição da subclasse;
- 
- O uso mais simples da herança é adicionar funcionalidade a uma classe existente.
  - Para ilustrar, veja um simples exemplo de uma classe base que gerencia contados:

In [2]:

```
1 class Contact:
2     all_contacts = []
3
4     def __init__(self, name, email):
5         self.name = name
6         self.email = email
7         Contact.all_contacts.append(self)
```

- Neste exemplo usamos o que chamamos de variável de classe;
  - A lista `all_contacts` é uma variável de classe porque é compartilhada por todos objetos da classe `Contact`.
- Agora imagine que existem contatos de fornecedores que fazemos pedidos;
  - Poderíamos pensar em adicionar um método **order** na classe `Contact`, mas isso faria que contatos familiares também tivessem esse método;
  - Vamos criar uma classe `Fornecedor` (*Supplier* em inglês) que além de herdar de `Contact`, também possui o método `order`:

In [3]:

```
1 class Supplier(Contact):
2     def order(self, pedido):
3         print("Pedido para '{}': "
4               "'{}'.format(self.name, pedido))
```

In [4]:

```
1 c1 = Contact('Filipe', 'filipe@gmail.com')
2 f1 = Supplier('Amazon', 'amazon@gmail.com')
```

- Note que o objeto `f1` possui nome e email pois herda da classe `Contact`:

In [5]:

```
1 print(c1.name, c1.email)
2 print(f1.name, f1.email)
```

```
Filipe filipe@gmail.com
Amazon amazon@gmail.com
```

- Perceba ainda que apenas o objeto `f1` é capaz de efetuar pedidos:

In [6]:

```
1 f1.order('Preciso de 2 camisas do palmeiras')
```

```
Pedido para 'Amazon': 'Preciso de 2 camisas do palmeiras'
```

In [7]:

```
1 c1.order('Preciso de 2 camisas do palmeiras')
```

```
-----  
-----  
AttributeError                                Traceback (most recent call  
last)  
<ipython-input-7-c139743b71d5> in <module>  
----> 1 c1.order('Preciso de 2 camisas do palmeiras')
```

AttributeError: 'Contact' object has no attribute 'order'

- Agora perceba como o atributo `all_contacts` é compartilhado por ambos objetos:

In [8]:

```
1 c1.all_contacts
```

Out[8]:

```
[<__main__.Contact at 0x7fc1c12d5d30>, <__main__.Supplier at 0x7fc1c12d5cc0>]
```

In [9]:

```
1 f1.all_contacts
```

Out[9]:

```
[<__main__.Contact at 0x7fc1c12d5d30>, <__main__.Supplier at 0x7fc1c12d5cc0>]
```

- Como ele é um atributo de classe, você pode acessá-lo diretamente pela classe:

In [10]:

```
1 Contact.all_contacts
```

Out[10]:

```
[<__main__.Contact at 0x7fc1c12d5d30>, <__main__.Supplier at 0x7fc1c12d5cc0>]
```

## Herdando de built-ins

- Um uso interessante desse tipo de herança é adicionar funcionalidade aos recursos built-ins do python;
- Para exemplo, podemos criar uma classe `ContactList` que adiciona um método para fazer busca de contatos em uma lista padrão do python;
  - Para tanto, herdaremos de `list`:
- Após isso, iremos usar a `ContactList` ao invés de uma lista padrão para iniciar nossa variável de classe `all_contacts` (linha 10 do código abaixo):

In [11]:

```
1 ▾ class ContactList(list):
2 ▾     def search(self, name):
3         matching_contacts = []
4 ▾     for contact in self:
5 ▾         if name in contact.name:
6             matching_contacts.append(contact)
7         return matching_contacts
8
9 ▾ class Contact():
10     all_contacts = ContactList()
11
12 ▾     def __init__(self, name, email):
13         self.name = name
14         self.email = email
15         Contact.all_contacts.append(self)
```

- Para testar nosso exemplo, vamos criar três contatos:

In [12]:

```
1 c1 = Contact('Fulano da Silva', 'fulano1@dominio.com')
2 c2 = Contact('Fulano Sousa', 'fulano2@dominio.com')
3 c3 = Contact('Ciclano Pereira', 'ciclano3@dominio.com')
```

- Agora vamos fazer uma busca por contatos com nome *Fulano*;

In [13]:

```
1 Contact.all_contacts.search('Fulano')
```

Out[13]:

```
[<__main__.Contact at 0x7fc1c1306f60>, <__main__.Contact at 0x7fc1c1306e48>]
```

- Bem, existem dois objetos com esse nome;
  - Vamos apresentá-los:

In [14]:

```
1 [c.name for c in Contact.all_contacts.search('Fulano')]
```

Out[14]:

```
['Fulano da Silva', 'Fulano Sousa']
```

- Note que a sintaxe built-in `[]` é equivalente a `list()`:

In [15]:

```
1 [] == list()
```

Out[15]:

True

- Assim sendo, quando fazemos *variavel = []*, estamos instanciando um objeto da classe *list*;
- Ainda, a classe *list* herda de *object*, como podemos ver abaixo:

In [16]:

```
1 isinstance([], object)
```

Out[16]:

True

- Como um segundo exemplo, vamos herdar agora da classe *dict*;
- Iremos adicionar uma funcionalidade para verificar qual é a chave mais longa do dicionário:

In [17]:

```
1 class LongNameDict(dict):
2     def longest_key(self):
3         longest = None
4         for key in self.keys():
5             if not longest or len(key) > len(longest):
6                 longest = key
7         return longest
```

In [18]:

```
1 longkeys = LongNameDict()
2 longkeys['hello'] = 1
3 longkeys['longest yet'] = 5
4 longkeys['hello2'] = 'world'
5 longkeys.longest_key()
```

Out[18]:

'longest yet'

## Sobrescrevendo métodos

- Além de adicionar funcionalidades ao utilizar herança, podemos também modificá-las;
  - Isto é, podemos herdar comportamentos e modificá-los;
- Por exemplo, imagine que queremos adicionar o atributo *phone\_number* nos contatos dos nossos amigos mais próximos;
  - Para tanto, podemos criar uma classe *Friend* que herda de *Contact* e sobrescreve o construtor:

In [19]:

```
1 class Friend(Contact):
2     def __init__(self, name, email, phone):
3         self.name = name
4         self.email = email
5         self.phone = phone
```

- Observe 2 problemas com o código acima:
  1. Repetimos código o que pode tornar o processo de manutenção do sistema ruim;
  2. Esquecemos de adicionar o contato do amigo na variável de classe `all_contacts`;
- O que nós realmente precisamos é executar o `__init__` da classe `Contact` e apenas adicionar o telefone do contato;
- Para tanto, podemos usar a palavra chave **super**, que nos permite invocar métodos da superclasse diretamente:

In [20]:

```
1 class Friend(Contact):
2     def __init__(self, name, email, phone):
3         super().__init__(name, email)
4         self.phone = phone
```

- Vamos criar um amigo, apenas para testar a classe:

In [21]:

```
1 friend1 = Friend('Beltrano Ferreira', 'beltrano@dominio.com', '2121-3131')
```

In [22]:

```
1 friend1.phone
```

Out[22]:

```
'2121-3131'
```

In [23]:

```
1 Contact.all_contacts
```

Out[23]:

```
[<__main__.Contact at 0x7fc1c1306f60>,
<__main__.Contact at 0x7fc1c1306e48>,
<__main__.Contact at 0x7fc1c1306f98>,
<__main__.Friend at 0x7fc1c0a7b048>]
```

## Herança Múltipla

- A herança múltipla é um assunto delicado.
  - Em teoria, é bem simples: uma subclasse que herda diretamente funcionalidades de mais uma superclasse;
  - Na prática, essa técnica não é tão usual e muitos experts não a recomendam;

- A maneira mais fácil de realizar herança múltipla é chamada de **mixin**;
  - Mixin é geralmente uma superclasse que só existe para adicionar funcionalidade a outras classes através da herança;
  - Por exemplo, vamos adicionar funcionalidades a nossa classe *Contact* para poder enviar e-mail para o *self.email*;

In [24]:

```
1 class MailSender:
2     def send_mail(self, message):
3         print('Enviando e-mail para ' + self.email)
4         # logica de envio de e-mail
```

- Podemos então criar uma classe que herda de *Contact* e *MailSender*:

In [25]:

```
1 class EmailableContact(Contact, MailSender):
2     pass
```

- Agora vamos testar a nossa classe híbrida:

In [26]:

```
1 e = EmailableContact("John Smith", "jsmith@example.net")
```

In [27]:

```
1 e.send_mail("Hello, test e-mail here")
```

Enviando e-mail para jsmith@example.net

- Observe que o objeto da classe *EmailableContact* também está sendo adicionado na lista *all\_contacts*:

In [30]:

```
1 Contact.all_contacts
```

Out[30]:

```
[<__main__.Contact at 0x7fc1c1306f60>,
 <__main__.Contact at 0x7fc1c1306e48>,
 <__main__.Contact at 0x7fc1c1306f98>,
 <__main__.Friend at 0x7fc1c0a7b048>,
 <__main__.EmailableContact at 0x7fc1c0a7b320>]
```

## Nota sobre envio de email em python

- Caso queira realmente enviar um e-mail, segue abaixo um breve tutorial onde o remetente tem uma conta do gmail.
  - Primeiro, acesse este [link \(https://myaccount.google.com/lesssecureapps?pli=1\)](https://myaccount.google.com/lesssecureapps?pli=1) e dê permissão para aplicações menos seguras;
  - A seguir, use o seguinte código:

In [ ]:

```
1 import smtplib
2 from email.mime.text import MIMEText
3 from email.mime.multipart import MIMEMultipart
4
5 email = 'seu_email@gmail.com'
6 senha = input('Entre com a senha:')
7 receptor = 'seu_email@gmail.com'
8 assunto = 'Ufrr python - P00'
9 conteudo = 'Aula de P00 - Testando o envio de um email'
10
11 msg = MIMEMultipart()
12 msg['From'] = email
13 msg['To'] = receptor
14 msg['Subject'] = assunto
15 msg.attach(MIMEText(conteudo, 'plain'))
16
17 servidor = smtplib.SMTP('smtp.gmail.com', 587)
18 servidor.starttls()
19 servidor.login(email, senha)
20 text = msg.as_string()
21 servidor.sendmail(email, receptor, text)
22 servidor.quit()
```

- **Créditos:** o código acima foi feito com base nesta [videoaula \(https://www.youtube.com/watch?v=YPiHBtddefI\)](https://www.youtube.com/watch?v=YPiHBtddefI) do canal PyTutorials;
- Para um tutorial um pouco mais completo, acesso este [link \(https://realpython.com/python-send-email/\)](https://realpython.com/python-send-email/) do site real python;

## Um pouco mais sobre herança múltipla

- Note que poderíamos ter usado composição ao invés de herança na classe EmailableContact:
  - A classe EmailableContact poderia ter um objeto MailSender ao invés de herdar dele, o que faz muito mais sentido;
- Herança Múltipla pode se tornar bastante confuso quando precisamos invocar métodos da superclasse, uma vez que existem múltiplas superclasses;
  - Para explorar um pouco mais sobre esse assunto, vamos criar uma classe que modela objetos detentores de endereço:

In [30]:

```
1 class AddressHolder:
2     def __init__(self, street, city, state, code):
3         self.street = street
4         self.city = city
5         self.state = state
6         self.code = code
```

- Vamos usar a classe acima com pai da nossa classe Friend;
- Entretanto, percebam que a classe Friend já herda de Contact e que usamos a palavra chave super para modificar um comportamento.

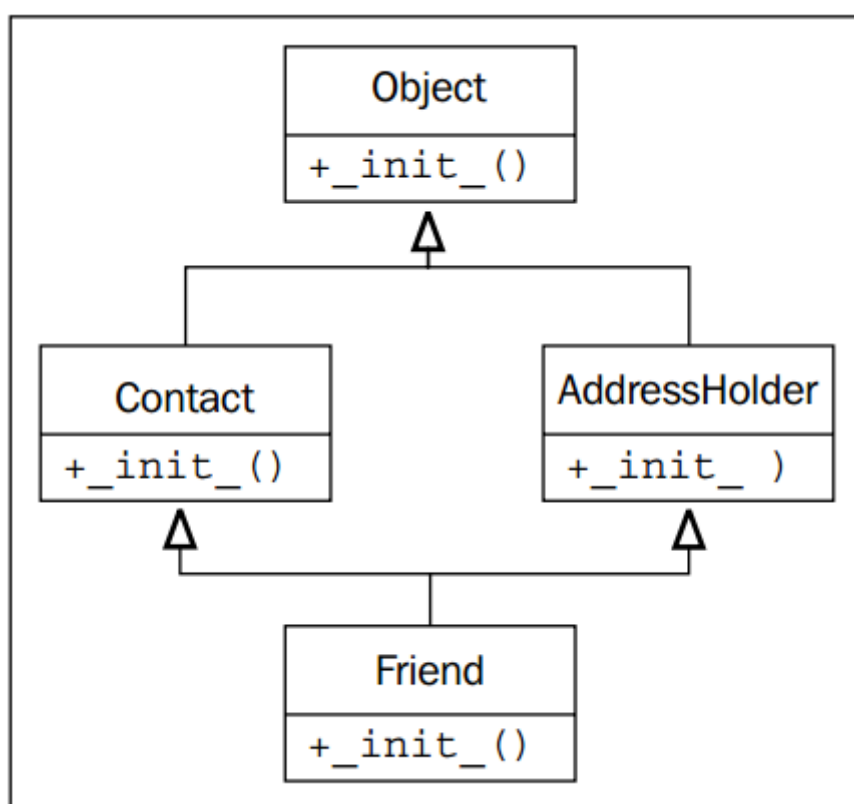


- Com herança múltipla, ao usar por exemplo, `super().__init__()` estamos nos referindo a qual das superclasses?
- Uma opção para tentar contornar esse problema é especificar a super classe e passar explicitamente o `self` como parâmetro no `__init__()` da superclasse, conforme segue:

In [38]:

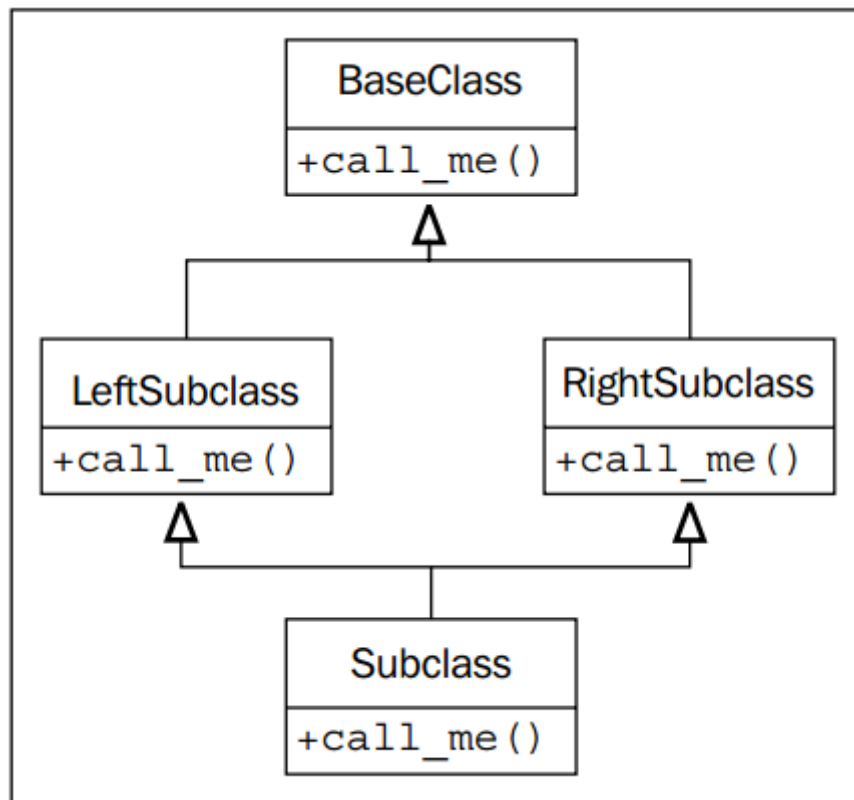
```
1 class Friend(Contact, AddressHolder):
2     def __init__(self, name, email, phone, street, city, state, code):
3         Contact.__init__(self, name, email)
4         AddressHolder.__init__(self, street, city, state, code)
5         self.phone = phone
```

- O código acima funciona, mas é perigoso.
- Veja o seguinte diagrama de classes:



- Observe que:
  - O método `__init__()` da classe Friends invoca o `__init__()` de Contact, que implicitamente invoca o `__init__()` da classe Object;
  - Subsequentemente, o método `__init__()` da classe Friends invoca o `__init__()` de AddressHolder, que também implicitamente invoca o `__init__()` da classe Object;
  - Assim, o `__init__()` da classe Object é chamado duas vezes;
  - Para este cenário não há problema algum, mas imagine que no lugar de Object está a classe Database e no `__init__()` do Database temos uma conexão com o banco de dados;
    - Faríamos a conexão duas vezes para cada requisição, o que pode ser desastroso;
- Vejamos um segundo exemplo hipotético que ilustra esse problema mais claramente:
  - Aqui temos uma classe base que possui um método chamado `call_me`;
  - As subclasses LeftSubClass e RightSubClass herdam diretamente de BaseClass e sobreescrevem o método `call_me`;

- A classe Subclass herda de LeftSubClass e RightSubClass e também sobrescreve *call\_me*;
- Isso é chamado de herança de diamante (diamond inheritance) devido à forma do diamante do diagrama de classes:



- Vamos converter esse diagrama em código:

In [39]:

```
1  class BaseClass:
2      num_base_calls = 0
3  def call_me(self):
4      print("Calling method on Base Class")
5      self.num_base_calls += 1
6
7  class LeftSubclass(BaseClass):
8      num_left_calls = 0
9  def call_me(self):
10     BaseClass.call_me(self)#linha importante
11     print("Calling method on Left Subclass")
12     self.num_left_calls += 1
13
14 class RightSubclass(BaseClass):
15     num_right_calls = 0
16 def call_me(self):
17     BaseClass.call_me(self)#linha importante
18     print("Calling method on Right Subclass")
19     self.num_right_calls += 1
20
21 class Subclass(LeftSubclass, RightSubclass):
22     num_sub_calls = 0
23 def call_me(self):
24     LeftSubclass.call_me(self)#linha importante
25     RightSubclass.call_me(self)#linha importante
26     print("Calling method on Subclass")
27     self.num_sub_calls += 1
```

- Vamos testar a nossa subclasse:

In [41]:

```
1  s = Subclass()
2  s.call_me()
3  print(s.num_sub_calls)
4  print(s.num_left_calls)
5  print(s.num_right_calls)
6  print(s.num_base_calls)
```

Calling method on Base Class  
Calling method on Left Subclass  
Calling method on Base Class  
Calling method on Right Subclass  
Calling method on Subclass

1  
1  
1  
2

- Aqui podemos ver claramente a classe BaseClass sendo chamada duas vezes;
- Apenas como mais uma demonstração dos problemas que isso pode causar, imagine que esse método está realizando um depósito em um banco;
- Podemos resolver isso usando o nosso **super**, que se encarregará de chamar o método da superclasse

apenas uma vez;

- Vamos reescrever nossa classe agora utilizando o super:

In [31]:

```
1  class BaseClass:
2      num_base_calls = 0
3  def call_me(self):
4      print("Calling method on Base Class")
5      self.num_base_calls += 1
6
7  class LeftSubclass(BaseClass):
8      num_left_calls = 0
9  def call_me(self):
10     super().call_me()#linha importante
11     print("Calling method on Left Subclass")
12     self.num_left_calls += 1
13
14 class RightSubclass(BaseClass):
15     num_right_calls = 0
16 def call_me(self):
17     super().call_me()#linha importante
18     print("Calling method on Right Subclass")
19     self.num_right_calls += 1
20
21 class Subclass(LeftSubclass, RightSubclass):
22     num_sub_calls = 0
23 def call_me(self):
24     super().call_me()#linha importante
25     print("Calling method on Subclass")
26     self.num_sub_calls += 1
```

- Agora vamos novamente o nosso teste e perceba que a classe base só será chamada uma vez:

In [32]:

```
1  s = Subclass()
2  s.call_me()
3  print(s.num_sub_calls)
4  print(s.num_left_calls)
5  print(s.num_right_calls)
6  print(s.num_base_calls)
```

```
Calling method on Base Class
Calling method on Right Subclass
Calling method on Left Subclass
Calling method on Subclass
```

```
1
1
1
1
```

Vale ressaltar como funcionou esse processo para que a classe base seja chamada apenas uma vez:

- Primeiro foi invocado `call_me` da SubClass que chamou `super().call_me()`, que se refere a `LeftSubClass.call_me()`.

- `LeftSubClass.call_me()`, por sua vez, chama `super().call_me()`, mas nesse caso, `super()` está se referindo a `RightSubclass.call_me()`;
  - Assim sendo, o `call_me()` da `BaseClass` na verdade não está sendo chamado na classe `LeftSubClass`;
  - Ao invés disso, ele é chamado na `RightSubClass`;

## Conjunto diferentes de argumentos

- Voltando para nossa classe `Friend`, temos uma chamada ao `__init__()` para ambas as superclasses;
  - Entretanto o conjunto de argumentos passados são diferentes para cada superclasse, conforme segue:

In [33]:

```
1 class Friend(Contact, AddressHolder):
2     def __init__(self, name, email, phone, street, city, state, code):
3         Contact.__init__(self, name, email) ## parâmetros diferentes
4         AddressHolder.__init__(self, street, city, state, code) ## parâmetros
5         self.phone = phone
```

- Como podemos gerenciar diferentes conjuntos de argumentos com o **super**?
  - Note que o que queremos é passar parâmetros adicionais nas subclasses;
  - Assim, precisamos projetar a superclasse para que ela possa ser estendida com novos parâmetros inesperados;
  - Para tanto, usaremos o **kwargs** na superclasse;
  - Antes de usá-lo, vamos ver um exemplo de uso do **kwargs** em uma função para somar valores de um dicionário:

In [33]:

```
1 def soma_valores(**kwargs):
2     result = 0
3     for arg in kwargs.values():
4         result += arg
5     return result
6
7 print(soma_valores(a=1, b=2, c=3, d=4))
```

10

Note que "a", "b", "c" e "d" se tornaram chaves do dicionário **kwargs**, enquanto que 1, 2, 3 e 4 se tornaram os valores desse dicionário;

- Isso quer dizer, que podemos acessar o valor 1 através do comando `kwargs["a"]`:

In [32]:

```
1 ▾ # esse código só funciona se passarmos o 'a' como argumento:
2 ▾ def exemplo_uso(**kwargs):
3     return kwargs["a"]
4
5 print(exemplo_uso(a=1))
```

1

Voltando a nossas classes, vamos projetá-las para receber parâmetros inesperados no construtor utilizando o `**kwargs`:

In [36]:

```
1 ▾ class Contact:
2     all_contacts = []
3
4 ▾     def __init__(self, name, email, **kwargs):
5         super().__init__(**kwargs)
6         self.name = name
7         self.email = email
8         self.all_contacts.append(self)
9
10 ▾ class AddressHolder:
11     def __init__(self, street='', city='', state='', code='', **kwargs):
12         super().__init__(**kwargs)
13         self.street = street
14         self.city = city
15         self.state = state
16         self.code = code
17
18 ▾ class Friend(Contact, AddressHolder):
19     def __init__(self, phone='', **kwargs):
20         super().__init__(**kwargs)
21         self.phone = phone
```

- Ao instanciar um objeto da classe `Friend`, devemos obrigatoriamente passar os atributos *name* e *email*
- Os outros atributos serão inicializados com uma string vazia "
- Veja abaixo dois objetos da classe `Friend`, onde `f1` é inicializado apenas com os atributos obrigatórios e `f2` é inicializado com todos os atributos possíveis:

In [53]:

```
1 f1 = Friend(name='Fulano de Tal', email='fulano@gmail.com')
```

In [39]:

```
1 ▾ f2 = Friend(name='Ciclano de Tal', email='ciclano@gmail.com', phone='2121-3131',
2     street='Ataide Teive', city='Boa Vista', state='Roraima',
3     code = '69300-000')
```

In [41]:

```
1 f2.state
```

Out[41]:

```
'Roraima'
```

## Nota sobre kwargs:

- `**kwargs` basicamente coleta todos os argumentos passados para o método que não foram listados explicitamente na lista de parâmetros.
- esses argumentos são armazenados em um dicionário chamado `kwargs` (você pode colocar qualquer outro nome se quiser);
- Quando chamamos um método diferente (por exemplo, `super().__init__()`) com `**kwargs`, descompacta-se o dicionário e são passados os resultados para o método como argumentos normais;
- Voltando para a classe `Friend`, a pergunta que fica é:
  - Quando instanciarmos um objeto da classe `Friend`, o que deve ser passado como parâmetro além do `phone`?
    - Para responder isso é importante disponibilizar os atributos possíveis no docstring, do contrário o usuário da classe ficará confuso:

In [ ]:

```
1 f = Friend(phone='2121-3131') # e o que mais?
```

Por fim, herança múltipla pode ser útil, mas em muitos casos é mais apropriado combinar classes usando composição ou um dos design patterns que veremos nas próximas aulas;

## Polimorfismo

- Como vimos na primeira aula, polimorfismo é a capacidade do objeto de uma subclasse ser referenciado como uma superclasse para que, dependendo da subclasse, posso haver comportamentos diferentes.
  - Em outras palavras, diferentes comportamentos podem ocorrer dependendo de qual subclasse está sendo usada, onde não existe a necessidade de sabermos explicitamente que subclasse é essa;
- Como um exemplo, imagine um programa que toca um arquivo de áudio;
- Um *player* precisará carregar um objeto *AudioFile* e tocá-lo;
  - Para tanto, deverá existir um método *play* na classe *AudioFile*;
- O player basicamente executará o seguinte comando:

In [ ]:

```
1 audio_file.play()
```

- No entanto, o processo de compactação e extração de um arquivo de áudio é muito diferente para diferentes tipos de arquivos.

- Os arquivos .wav são armazenados descompactados, enquanto os arquivos .mp3, .wma e .ogg possuem algoritmos de compactação totalmente diferentes.
- Podemos usar polimorfismo e herança para facilitar o processo:
  - Cada tipo de arquivo pode ser uma subclasse de `AudioFile`;
  - Cada subclasse deve ter o método `play`, mas esse método será diferente em cada subclasse;
  - O objeto media player nunca saberá que a subclasse do `AudioFile` está sendo referenciada;
  - O objeto media player apenas irá chamar o método `play()` e o polimorfismo irá se encarregar dos detalhes de como realizar o play;
- Vamos codificar isso:

In [46]:

```

1  class AudioFile:
2      def __init__(self, filename):
3          if not filename.endswith(self.ext):
4              raise Exception("Invalid file format")
5          self.filename = filename
6
7  class MP3File(AudioFile):
8      ext = "mp3"
9      def play(self):
10         print("tocando {} como mp3".format(self.filename))
11
12 class WavFile(AudioFile):
13     ext = "wav"
14     def play(self):
15         print("tocando {} como wav".format(self.filename))
16
17 class OggFile(AudioFile):
18     ext = "ogg"
19     def play(self):
20         print("tocando {} como ogg".format(self.filename))

```

- Todos os arquivos de áudio são verificados para garantir que uma extensão válida seja fornecida na inicialização;
- Agora observe como o método `__init__` da super classe `AudioFile` fez para acessar a extensão do arquivo (`self.ext`);
  - Isso é polimorfismo. Se o nome do arquivo não terminar com o extensão correta, será gerada uma exceção (esse assunto será tratado na próxima aula);
- Além disso, cada subclasse de `AudioFile` implementa o método `play()` de uma maneira diferente;
  - Isso também é polimorfismo;
  - O *media player* pode usar exatamente o mesmo código (`audio_file.play()`) para reproduzir um arquivo de áudio, independentemente do tipo;
  - Não importa qual subclasse de `AudioFile`.
- Vamos testar o código:

In [10]:

```

1  ogg = OggFile("myfile.ogg")
2  ogg.play()

```

tocando myfile.ogg como ogg



In [11]:

```
1 mp3 = MP3File("myfile.mp3")
2 mp3.play()
```

tocando myfile.mp3 como mp3

In [12]:

```
1 not_an_mp3 = MP3File("myfile.ogg")
```

-----  
-----  
Exception Traceback (most recent call last)

<ipython-input-12-80a298705989> in <module>

----> 1 not\_an\_mp3 = MP3File("myfile.ogg")

<ipython-input-9-0714db3a359a> in \_\_init\_\_(self, filename)

2 def \_\_init\_\_(self, filename):

3 if not filename.endswith(self.ext):

----> 4 raise Exception("Invalid file format")

5 self.filename = filename

6

Exception: Invalid file format

## Duck Typing

- Como foi visto na primeira aula, **Duck Typing** em Python nos permite usar qualquer objeto que forneça o comportamento necessário sem forçá-lo a ser uma subclasse;
- O exemplo a seguir não estende o da classe AudioFile, mas pode ser tocado no media player pois usa a mesma interface:

In [ ]:

```
1 class FlacFile:
2     def __init__(self, filename):
3         if not filename.endswith(".flac"):
4             raise Exception("Invalid file format")
5         self.filename = filename
6     def play(self):
7         print("playing {} as flac".format(self.filename))
```

## Notas:

- O polimorfismo é uma das razões mais importantes para usar a herança em muitos contextos orientados a objetos.
- Como qualquer objeto que forneça a interface correta pode ser usado de forma intercambiável em Python, reduz-se a necessidade superclasses pensadas para realizar polimorfismo.
- A herança ainda pode ser útil para o compartilhamento de código, mas, se tudo o que está sendo compartilhado é a interface pública, o Duck Typing é a técnica apropriada em python.
  - Isso reduz a necessidade de herança simples e múltipla;
  - Normalmente herança múltipla pode ser simulada com Duck Typing;

# Classes Abstratas

- Apesar do Duck Typing ser bastante útil, nem sempre é fácil saber com antecedência se uma classe vai cumprir o protocolo estabelecido na "interface".
- Classes abstratas (Abstract Base Class - ABCs) definem um conjunto de métodos e propriedades que uma subclasse deve implementar para ser considerada uma instância duck typing;
  - Uma classe torna-se uma subclasse da ABC ao fornecer a implementação de todos os métodos abstratos definidos na ABC.

## Usando uma classe abstrata

- A maioria das classes abstratas que já existem na biblioteca padrão do python estão no módulo *collections*;
  - Vamos inspecionar quais são os métodos abstratos da classe abstrata Container:

In [4]:

```
1 from collections import Container
2 Container.__abstractmethods__
```

Out[4]:

```
frozenset({'__contains__'})
```

- Como podemos ver, a classe Container só tem o método abstrato `__contains__`;
- Vamos usar o comando `help` para verificar a assinatura deste método:

In [5]:

```
1 help(Container.__contains__)
```

Help on function `__contains__` in module `collections.abc`:

```
__contains__(self, x)
```

- A assinatura não é tão informatimava, mas podemos concluir intuitivamente que o `x` é o valor passado para verificar se ele está dentro do Container;
- Esse método é implementado, por exemplo, pelas classes *list*, *str* e *dict* para indicar se um dado valor está ou não presente na estrutura de dados;
- Entretanto, apenas para fins didáticos, vamos implementar um Container bem simples que verifica se um dado valor é do tipo inteiro e se ele é par:

In [6]:

```
1 class OddContainer:
2     def __contains__(self, x):
3         if isinstance(x, int) and x%2==0:
4             return True
5         return False
```

- Vamos testar o método `__contains__` do nosso Container de valores pares:

In [8]:

```
1 odd_container = OddContainer()
2 odd_container.__contains__(2)
```

Out[8]:

True

In [9]:

```
1 odd_container.__contains__('teste')
```

Out[9]:

False

- Agora note que ao implementar o método `__contains__` nossa classe passar a ser um Container (duck typing):

In [10]:

```
1 isinstance(odd_container, Container)
```

Out[10]:

True

- Nosso OddContainer é uma subclasse de Container (duck typing):

In [ ]:

```
1 issubclass(OddContainer, Container)
```

- Seguindo a filosofia do *duck typing*, se a classe tem o método `__contains__`, então ela é um Container ;-)
- Isso torna duck typing uma alternativa em tanto se comparada ao polimorfismo clássico que depende da herança;

## Nota

- Algo interessante é que do Container ABC é que a palavra chave **in** do python nada mais é do que um *syntax sugar* do método `__contains__`;
  - Assim sendo, qualquer classe que implementar o `__contains__` é um Container e, portanto, pode usar o **in** conforme segue:

In [11]:

```
1 10 in odd_container
```

Out[11]:

True

In [12]:

```
1 1 in odd_container
```

Out[12]:

False

In [13]:

```
1 'uma string qualquer' in odd_container
```

Out[13]:

False

## Criando nossas próprias ABCs

- Como vimos anteriormente, não é necessário criar uma ABC para permitir o uso do duck typing;
- No entanto, imagine que queremos criar um media player com plugins de terceiros.
- É recomendável criar uma ABC para documentar qual API os plug-ins de terceiros devem fornecer.
  - Vejamos como fazer isso:

In [15]:

```
1 import abc
2
3 class MediaLoader(metaclass=abc.ABCMeta):
4     @abc.abstractmethod
5     def play(self):
6         pass
7
8     @abc.abstractproperty
9     def ext(self):
10         pass
11
12     @classmethod
13     def __subclasshook__(cls, C):
14         if cls is MediaLoader:
15             attrs = set(dir(C))
16             if set(cls.__abstractmethods__) <= attrs:
17                 return True
18
19             return NotImplemented
```

- Este é um exemplo complicado, onde alguns recursos serão explicados posteriormente;
  - Nota: você não precisa entender tudo isso para ter uma ideia de como criar sua própria ABC.

- De modo bem superficial, ao usar `metaclass=abc.ABCMeta` na linha 3, estamos dando à classe `MediaLoader` poderes de superclasse;
- A seguir usamos os decarators `@abc.abstractmethod` e `@abc.abstractproperty` para impor que subclasses desta superclasse deve fornecer implementação para esse método/propriedade;
  - Veja abaixo o que acontece se não fornecermos implementações para esse método e propriedade:

In [17]:

```
1 class Wav(MediaLoader):
2     pass
```

In [18]:

```
1 x = Wav()
```

```
-----
-----
TypeError                                 Traceback (most recent call
last)
<ipython-input-18-3bef4301f672> in <module>
----> 1 x = Wav()
```

**TypeError:** Can't instantiate abstract class Wav with abstract methods ext, play

- Como a classe `Wav` não fornece as implementações dos itens abstratos, então ela não pode ser instanciada;
  - Nesse caso, ao não fornecer tais implementações `Wav` se torna uma classe abstrata também e **classes abstratas não podem ser instanciadas**;
- Agora, como contraexemplo, vamos criar uma classe que implementa o método e a propriedade abstrata:

In [19]:

```
1 class Ogg(MediaLoader):
2     ext = '.ogg'
3
4     def play(self):
5         pass
```

In [20]:

```
1 o = Ogg()
```

- Voltando a nossa classe abstrata `MediaLoader`, vamos analisar o método `__subclasshook__`;
  - Esse é o método que informa que qualquer classe que fornecer uma implementação concreta de todos os itens abstratos da classe `MediaLoader` é uma subclasse de `MediaLoader`, **mesmo se a subclasse não herdar de MediaLoader**;
- Linguagens orientadas a objetos mais comuns têm uma clara separação entre a interface e a implementação de uma classe.
  - Por exemplo, algumas linguagens de programação fornecem uma palavra-chave **interface** que nos permite definir os métodos que uma classe deve ter sem nenhuma implementação.
  - Nesse tipo de linguagem, normalmente uma classe abstrata é aquela que fornece uma interface e uma implementação concreta de alguns métodos, mas não de todos os métodos.

- Os ABCs do Python ajudam a fornecer a funcionalidade das interfaces sem comprometer os benefícios do duck typing;

## Um pouco mais sobre a classe **MediaLoader**

**@classmethod** é um decarator que para marcar um método como método de classe;

- Isso significa que o método pode ser invocado diretamente pela classe, ao invés de por uma objeto instanciado pela classe;
- Outras linguagens de programação como o java chamam **@classmethod** de métodos estáticos;

**def \_\_subclasshook\_\_(cls, C):**

- Define o método de classe **\_\_subclasshook\_\_(cls, C)**, que é um método especial invocado pelo interpretador do Python para responder se uma determinada classe C é uma subclasse desta classe.

**if cls is MediaLoader:**

- Isso previne, por exemplo, que a classe Wav seja interpretada como a superclasse da classe Ogg;

**attrs = set(dir(C))**

- Tudo o que essa linha faz é obter o conjunto de métodos e propriedades que a classe possui, incluindo quaisquer classes pai em sua hierarquia de classes;

**if set(cls.abstractmethods) <= attrs**

- Esta linha usa a notação de conjunto para verificar se o conjunto de métodos abstratos nesta classe foram fornecidos na classe candidata.
- Observe que não é checado se eles foram implementados, mas apenas se eles estão lá.
  - Isso porque uma subclasse desta classe pode também ser abstrata;

**return True**

- Se todos os métodos abstratos foram fornecidos pela subclasse, então a classe candidata é uma subclasse e será retornado True;
  - Observe que o método pode retornar True, False ou NotImplemented, onde True ou False indicam se a classe é ou não uma subclasse. NotImplemented significa que a classe deixou de implementar um ou mais itens abstratos como métodos ou propriedades;

## Estudo de Caso

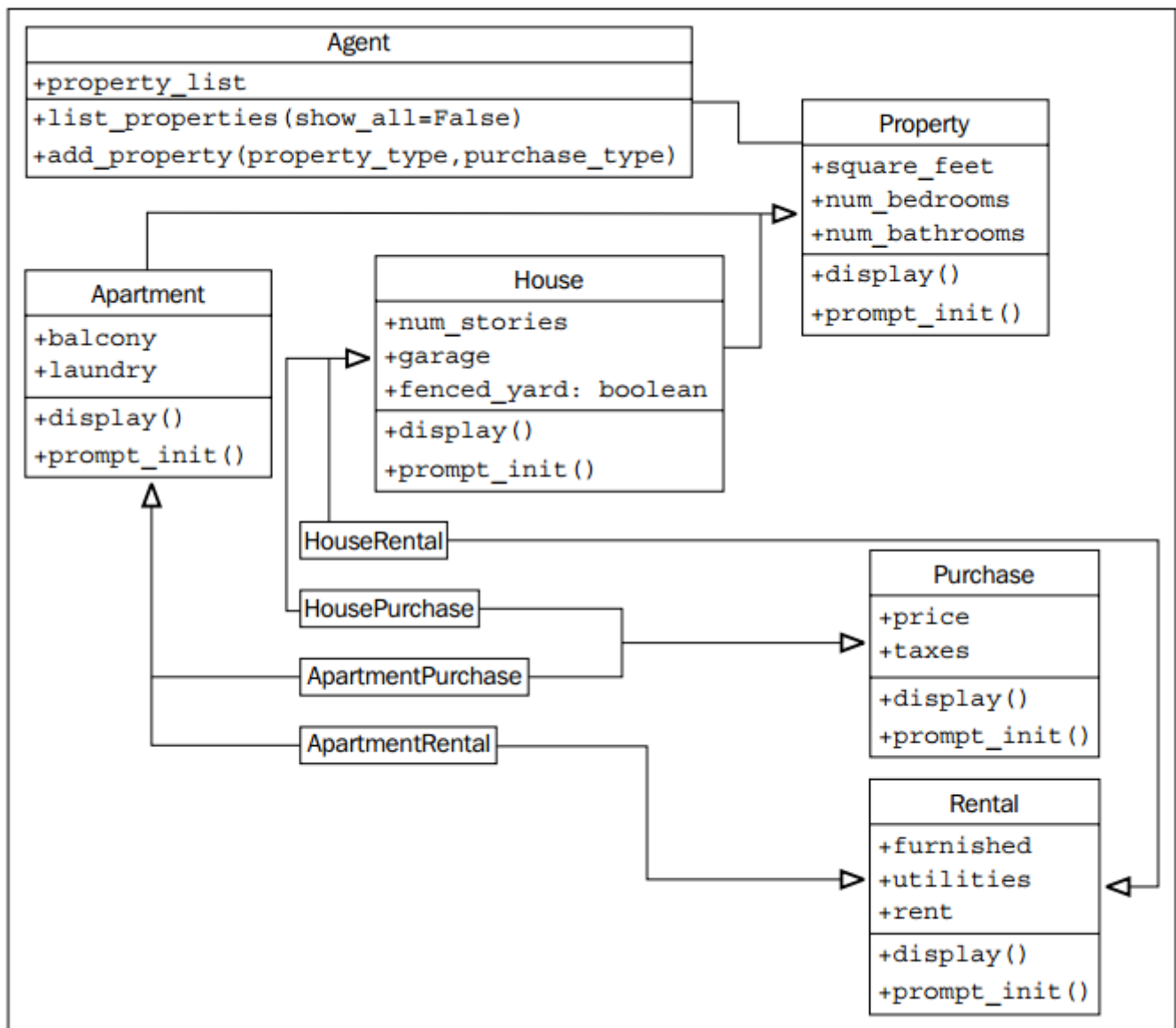
- Para colocar tudo o que aprendemos hoje em prática em um caso de uso;
- Modelaremos uma aplicação que permite que agentes gerenciem propriedades disponíveis para venda e aluguel;
- Existirão dois tipos de propriedades: apartamentos e casas;
- O agente poderá:

- fornecer detalhes relevantes sobre novas propriedades;
- listar todas as propriedades disponíveis;
- marcar uma propriedade como vendida ou alugada;
- Por fins de simplicidade, não nos preocuparemos em editar detalhes da propriedade ou reativar uma propriedade depois de vendida.
- Note que na nossa descrição de requisitos, foram mencionados alguns substantivos;
  - Lembre-se que substantivos são potenciais classes em projetos orientados a objetos;
- Nesta aula estudamos herança, então vamos pensar em formas de usá-la;
- Casa (*House*) e Apartamento (*Apartment*) são tipos de Propriedades (*Property*);
  - Nesse sentido, Propriedade é uma potencial superclasse para Casa e Apartamento;
- Agora vamos forçar um pouco a barra para usar herança nas classes Aluguel (*Rental*) e Compra (*Purchase*);
  - Criaremos classes separadas como por exemplo *HouseRental* e *HousePurchase* e usaremos herança múltipla, fazendo com que *HouseRental* herde de *House* e *Rental*;
    - Faremos analogamente para *HousePurchase*, *ApartmentRental*, *ApartmentPurchase*;
- Observe que poderíamos criar um projeto menos "desajeitado" usando composição e associações para não ter que usar principalmente herança múltipla;
- Entretanto, queremos praticar os conceitos aprendidos nesta aula ;-)
- Em relação aos atributos, a classe *Property* deve ter:
  - *square\_footage*: relacionado ao tamanho da propriedade;
  - *number\_of\_bedrooms*: número de quartos;
  - *number\_of\_bathrooms*: número de banheiros;
  - etc (por fins de simplicidade vamos parar por aqui).
- Já a classe *House* deve ter:
  - *number\_of\_stories*: número de andares;
  - *garage*: se tem uma garagem e ela é dentro do quintal, fora ou se não tem;
  - *fenced\_yard*: se o quintal é cercado;
- A classe *Apartment* deve ter:
  - *balcony*: se tem ou não uma varanda;
  - *laundry*: se a lavanderia fica no apartamento (ensuite), ou fora com sistema de pagamento com moedas (coin laundry) ou se fica fora sem esse sistema de moedas (off-site);

Note que ambas as propriedades precisarão de um método para mostrar as suas características;

- A classe *RentalProperty* precisará de atributos para:
  - *rent*: atributo para guardar o valor do aluguel;
  - *furnished*: para informar se a propriedade é ou não mobiliada;
  - *utilities*: contas a pagar (água, luz, internet e etc);
- Já a classe *PurchaseProperty* precisará de:
  - *purchase\_price*: valor do imóvel;
  - *taxes*: valor das taxas anuais;
- Usaremos o método *display()* para apresentar as informações relevantes das classes;
- Finalmente, precisaremos de uma classe *Agent*, que possui:
  - *property\_list*: lista de propriedades que ele agencia;
- Além disso, o *Agent* precisará de um método para listar propriedades e outro para adicionar novas propriedades.

- Vejamos o diagrama de classes do nosso projeto:



Partindo para o código, vamos começar pela classe **Property**:



In [3]:

```
1  ▾ class Property:
2  ▾     def __init__(self, square_feet='', beds='', baths='', **kwargs):
3         super().__init__(**kwargs)
4         self.square_feet = square_feet
5         self.beds = beds
6         self.baths = baths
7
8  ▾     def display(self):
9         print("DETALHES DA PROPRIEDADE")
10        print("=====")
11        print("Tamanho em pés quadrados: {}".format(self.square_feet))
12        print("Número de quartos: {}".format(self.beds))
13        print("Número de banheiros: {}\n".format(self.baths))
14
15  ▾     def prompt_init():#método de classe (método estático)
16  ▾         return dict(square_feet=input("Entre com o tamanho em pés quadrados:"),
17                        beds=input("Entre com o número de quartos:"),
18                        baths=input("Entre com o número de banheiros:"))
19     prompt_init = staticmethod(prompt_init)
```

- Alguns comentários sobre a classe acima:
  - Colocamos o `**kwargs`, porque já sabemos que vamos usar herança múltipla;
  - Colocamos também `super().__init__(**kwargs)` em caso de não sermos a última chamada na cadeia de herança múltipla;
  - como `prompt_init()` é um método de classe, ele não tem o argumento `self`;
  - o `prompt_init()` cria um dicionário com os valores que serão passados para o `__init__`

A classe `Apartment` vai estender `Property`:

In [9]:

```
1  class Apartment(Property):
2      valid_laundries = ("coin", "ensuite", "none")
3      valid_balconies = ("yes", "no", "solarium")
4
5      def __init__(self, balcony='', laundry='', **kwargs):
6          super().__init__(**kwargs)
7          self.balcony = balcony
8          self.laundry = laundry
9
10     def display(self):
11         super().display()
12         print("DETALHES DO APARTAMENTO")
13         print("lavanderia: %s" % self.laundry)
14         print("possui varanda: %s" % self.balcony)
15
16     def prompt_init():
17         parent_init = Property.prompt_init()
18         laundry = ''
19         while laundry.lower() not in Apartment.valid_laundries:
20             laundry = input("Como é a lavanderia da "
21                             "propriedade ? ({}).format(
22                                 ", ".join(Apartment.valid_laundries)))
23         balcony = ''
24         while balcony.lower() not in Apartment.valid_balconies:
25             balcony = input("A propriedade tem uma varanda? "
26                             "({}).format(", ".join(Apartment.valid_balconies)
27         parent_init.update({
28             "laundry": laundry,
29             "balcony": balcony
30         })
31         return parent_init
32     prompt_init = staticmethod(prompt_init)
```

- Sobre a classe acima, observe que o método `display` e `__init__` chamam os métodos de seus pais usando o `super()` para garantir que o imóvel será inicializado corretamente;
- O método `prompt_init` da classe `Apartment` está adicionando novos valores aos da classe `Property`;
  - Um problema deste método é que temos dois loops praticamente idênticos para validar as variáveis `laundry` e `balcony`;
  - Que tal se criarmos uma função capaz de validar variáveis:

In [7]:

```
1  def get_valid_input(input_string, valid_options):
2      input_string += " ({}).format(", ".join(valid_options))
3      response = input(input_string)
4      while response.lower() not in valid_options:
5          response = input(input_string)
6      return response
```

- Antes de mais nada, vamos testar essa função:

In [8]:

```
1 get_valid_input("como é a lavanderia?", ("coin", "ensuite", "none"))
```

```
como é a lavanderia? (coin, ensuite, none) hi
como é a lavanderia? (coin, ensuite, none) COIN
```

Out[8]:

```
'COIN'
```

- Agora vamos atualizar o método *prompt\_init* da classe *Apartment* para que ele fique mais fácil de ler e manter:

In [10]:

```
1 class Apartment(Property):
2     valid_laundries = ("coin", "ensuite", "none")
3     valid_balconies = ("yes", "no", "solarium")
4
5     def __init__(self, balcony='', laundry='', **kwargs):
6         super().__init__(**kwargs)
7         self.balcony = balcony
8         self.laundry = laundry
9
10    def display(self):
11        super().display()
12        print("DETALHES DO APARTAMENTO")
13        print("lavanderia: %s" % self.laundry)
14        print("possui varanda: %s" % self.balcony)
15
16    def prompt_init():
17        parent_init = Property.prompt_init()
18        laundry = get_valid_input("Como é a lavanderia da "
19                                "propriedade ?", Apartment.valid_laundries)
20        balcony = get_valid_input("A propriedade tem uma varanda? ",
21                                Apartment.valid_balconies)
22        parent_init.update({
23            "laundry": laundry,
24            "balcony": balcony
25        })
26        return parent_init
27    prompt_init = staticmethod(prompt_init)
```

- Agora vamos para a classe *House*:

In [11]:

```
1  ▾ class House(Property):
2      valid_garage = ("attached", "detached", "none")
3      valid_fenced = ("yes", "no")
4
5  ▾      def __init__(self, num_stories='', garage='', fenced='', **kwargs):
6          super().__init__(**kwargs)
7          self.garage = garage
8          self.fenced = fenced
9          self.num_stories = num_stories
10
11 ▾      def display(self):
12          super().display()
13          print("DETALHES DA CASA")
14          print("Número de andares: {}".format(self.num_stories))
15          print("garagem: {}".format(self.garage))
16          print("quintal cercado: {}".format(self.fenced))
17
18 ▾      def prompt_init():
19          parent_init = Property.prompt_init()
20          fenced = get_valid_input("O quintal é cercado ",
21                                  House.valid_fenced)
22          garage = get_valid_input("Tem garagem ",
23                                   House.valid_garage)
24          num_stories = input("A casa tem quantos andares? ")
25
26 ▾          parent_init.update({
27              "fenced": fenced,
28              "garage": garage,
29              "num_stories": num_stories
30          })
31          return parent_init
32      prompt_init = staticmethod(prompt_init)
```

- Vejamos agora a classe Purchase:

In [21]:

```
1  ▾ class Purchase:
2  ▾      def __init__(self, price='', taxes='', **kwargs):
3          super().__init__(**kwargs)
4          self.price = price
5          self.taxes = taxes
6
7  ▾      def display(self):
8          super().display()
9          print("DETALHES DA COMPRA")
10         print("preço de compra: {}".format(self.price))
11         print("taxas anuais: {}".format(self.taxes))
12
13 ▾      def prompt_init():
14 ▾          return dict(
15              price=input("Qual o preço de venda? "),
16              taxes=input("Qual o valor das taxas anuais? ")
17          )
18      prompt_init = staticmethod(prompt_init)
```

Agora a classe Rental:

In [15]:

```
1 class Rental:
2     def __init__(self, furnished='', utilities='', rent='', **kwargs):
3         super().__init__(**kwargs)
4         self.furnished = furnished
5         self.rent = rent
6         self.utilities = utilities
7
8     def display(self):
9         super().display()
10        print("DETALHES DO ALUGUEL")
11        print("aluguel: {}".format(self.rent))
12        print("contas: {}".format(self.utilities))
13        print("mobiliada: {}".format(self.furnished))
14
15    def prompt_init():
16        return dict(
17            rent=input("Qual o valor mensal do aluguel? "),
18            utilities=input("Qual o valor das contas (água, luz, etc.)? "),
19            furnished = get_valid_input("A propriedade é mobiliada? ",
20                                      ("yes", "no")))
21    prompt_init = staticmethod(prompt_init)
```

- Usando herança múltipla vamos criar a classe HouseRental:

In [16]:

```
1 class HouseRental(Rental, House):
2     def prompt_init():
3         init = House.prompt_init()
4         init.update(Rental.prompt_init())
5         return init
6     prompt_init = staticmethod(prompt_init)
```

- Note que como as duas superclasses de HouseRental usam apropriadamente o super em seus métodos, nem precisamos criar o método `__init__` e `display`;
- Isso não é o caso do `prompt_init`, uma vez que ele é um método de classe e, portanto, não usa o super;
- Agora vamos testar nossa classe HouseRental:

In [17]:

```
1 init = HouseRental.prompt_init()
```

```
Entre com o tamanho em pés quadrados:200
Entre com o número de quartos:5
Entre com o número de banheiros:3
O quintal é cercado (yes, no) yes
Tem garagem (attached, detached, none) attached
A casa tem quantos andares? 2
Qual o valor mensal do aluguel? 2000
Qual o valor das contas (água, luz, etc.)? 400
A propriedade é mobiliada? (yes, no) yes
```

Podemos passar o dicionário `init` como parâmetro para o construtor de um objeto da classe HouseRental:

In [18]:

```
1 house = HouseRental(**init)
2 house.display()
```

#### DETALHES DA PROPRIEDADE

=====

Tamanho em pés quadrados: 200

Número de quartos: 5

Número de banheiros: 3

#### DETALHES DA CASA

Número de andares: 2

garagem: attached

quintal cercado: yes

#### DETALHES DO ALUGUEL

aluguel: 2000

contas: 400

mobiliada: yes

### Nota sobre ordem das classes ao fazer ao realizar a herança múltipla

- A ordem das classes herdadas no exemplo anterior é importante;
- Se nós tivéssemos escrito "class HouseRental(House, Rental)" ao invés de "class HouseRental(Rental, House)", o método *display* não teria invocado o *Rental.display()*;
  - Na linha 9 da classe *Rental*, fizemos a chamada ao *super().display()*, para que quando um objeto da classe *HouseRental* invoque o *Rental.display()* através da herança múltipla, ele seja encaminhado para o *super* da classe *House* que, por sua vez, invoca o *display* da classe *Property*;
  - Dessa forma, serão chamados o *super* da classe *Rental* e da classe *House*;
- Podemos agora criar o restante de nossas classes que também usam herança múltipla:

In [19]:

```
1 class ApartmentRental(Rental, Apartment):
2     def prompt_init():
3         init = Apartment.prompt_init()
4         init.update(Rental.prompt_init())
5         return init
6     prompt_init = staticmethod(prompt_init)
```

In [22]:

```
1 class ApartmentPurchase(Purchase, Apartment):
2     def prompt_init():
3         init = Apartment.prompt_init()
4         init.update(Purchase.prompt_init())
5         return init
6     prompt_init = staticmethod(prompt_init)
```

In [23]:

```
1 ▼ class HousePurchase(Purchase, House):
2 ▼     def prompt_init():
3         init = House.prompt_init()
4         init.update(Purchase.prompt_init())
5         return init
6     prompt_init = staticmethod(prompt_init)
```

- Finalmente, vamos criar nossa classe Agent;
- Observe que para adicionar propriedade precisamos primeiramente saber o tipo da propriedade e se ela está a venda ou para alugar;
  - Uma vez que isso estiver determinado, basta chamarmos o método prompt\_init da classe apropriada
- Podemos fazer isso usando um menu:

In [24]:

```
1 ▼ class Agent:
2 ▼     def __init__(self):
3         self.property_list = []
4
5 ▼     def display_properties(self):
6 ▼         for property in self.property_list:
7             property.display()
8
9 ▼     type_map = {
10         ("house", "rental"): HouseRental,
11         ("house", "purchase"): HousePurchase,
12         ("apartment", "rental"): ApartmentRental,
13         ("apartment", "purchase"): ApartmentPurchase
14     }
15
16 ▼     def add_property(self):
17 ▼         property_type = get_valid_input(
18             "What type of property? ",
19             ("house", "apartment")).lower()
20 ▼         payment_type = get_valid_input(
21             "What payment type? ",
22             ("purchase", "rental")).lower()
23
24         PropertyClass = self.type_map[(property_type, payment_type)]
25         init_args = PropertyClass.prompt_init()
26         self.property_list.append(PropertyClass(**init_args))
```

- No método add\_property o que fizemos foi verificar qual é a classe apropriada consultado o dicionário type\_map e depois instanciamos o objeto usando a variável PropertyClass da classe apropriada;
- Não sabemos exatamente com qual classe estamos lidando e usamos o polimorfismo para chamar o método prompt\_init para pegar um dicionário com os dados da propriedade e passar para o construtor;
- Depois usamos a sintaxe de keyword argument para converter um dicionário em argumentos do construtor de um novo objeto que carrega os dados corretos;
- Abaixo, vamos testar nossa classe Agent:

In [25]:

```
1 agent = Agent()
```

In [26]:

```
1 agent.add_property()
```

```
What type of property? (house, apartment) apartment
What payment type? (purchase, rental) purchase
Entre com o tamanho em pés quadrados:200
Entre com o número de quartos:2
Entre com o número de banheiros:2
Como é a lavanderia da propriedade ? (coin, ensuite, none) ensuite
A propriedade tem uma varanda? (yes, no, solarium) yes
Qual o preço de venda? 100000
Qual o valor das taxas anuais? 200
```

In [27]:

```
1 agent.add_property()
```

```
What type of property? (house, apartment) house
What payment type? (purchase, rental) rental
Entre com o tamanho em pés quadrados:100
Entre com o número de quartos:3
Entre com o número de banheiros:4
O quintal é cercado (yes, no) yes
Tem garagem (attached, detached, none) none
A casa tem quantos andares? 2
Qual o valor mensal do aluguel? 2000
Qual o valor das contas (água, luz, etc.)? 500
A propriedade é mobiliada? (yes, no) yes
```



In [28]:

```
1 agent.display_properties()
```

DETALHES DA PROPRIEDADE

=====

Tamanho em pés quadrados: 200

Número de quartos: 2

Número de banheiros: 2

DETALHES DO APARTAMENTO

lavanderia: ensuite

possui varanda: yes

DETALHES DA COMPRA

preço de compra: 100000

taxas anuais: 200

DETALHES DA PROPRIEDADE

=====

Tamanho em pés quadrados: 100

Número de quartos: 3

Número de banheiros: 4

DETALHES DA CASA

Número de andares: 2

garagem: none

quintal cercado: yes

DETALHES DO ALUGUEL

aluguel: 2000

contas: 500

mobiliada: yes

## Exercício Avaliativo

- Faça os exercícios da seção 10.7 e 10.9, 11.4 e 11.6 da apostila **py14** da caelum. Envie as soluções pelo SIGAA. As soluções podem ser enviadas através de um arquivo compactado ou arquivo com o link do github com seu código;