



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

## Disclaimer

Esta aula é uma adaptação do capítulo 2 do livro:

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

Nesta aula aprenderemos:

- Como criar classes e instanciar objetos em python;
- Como adicionar atributos e comportamentos a objetos em python;
- Como organizar classes em pacotes e módulos;
- Como sugerir que usuários não poluam nossos dados;

## Criando Classes em Python

- Python é reconhecidamente uma linguagem de programação limpa.
- Seguindo essa linha, para criar uma classe vazia basta:

In [1]:

```
1 class MyFirstClass:
2     pass
```

- Observe que nomes de classes seguem o padrão **CamelCase**. Para mais informações sobre o estilo python consulte o [PEP 8 \(https://www.python.org/dev/peps/pep-0008/\)](https://www.python.org/dev/peps/pep-0008/);
- Vamos instanciar dois objetos (*a* e *b*) do nosso exemplo simplório de classe.
- Quando impressos os objetos, são apresentados qual classe eles pertencem e em qual endereço de memória eles estão alocados.
  - Observe que os endereços são diferentes o que mostra que os objetos são diferentes.

In [2]:

```
1 a = MyFirstClass()
2 b = MyFirstClass()
3
4 print(a)
5 print(b)
```

<\_\_main\_\_.MyFirstClass object at 0x7fc1d0532588>

<\_\_main\_\_.MyFirstClass object at 0x7fc1d0596278>

## Adicionando Atributos

- Como python é uma linguagem dinâmica, podemos criar atributos diretamente usando a notação com ponto (**dot notation**)
  - `< object > . < attribute > = < value >`
  - No python os valores podem ser qualquer coisa (um tipo built-in, outro objeto, um função ou até uma classe)
- Exemplo simples:

In [3]:

```
1 class Point:
2     pass
3
4 p1 = Point()
5 p2 = Point()
```

In [4]:

```
1 p1.x = 5
2 p1.y = 4
3
4 p2.x = 3
5 p2.y = 6
```

In [5]:

```
1 print(p1.x, p1.y)
2 print(p2.x, p2.y)
```

5 4  
3 6

## Exemplo prático de uso de atributos

- Programação orientada a objetos é muito sobre interação entre objetos.
- Estamos interessados em invocar ações que causam mudanças nos atributos.
  - Para tanto, podemos adicionar comportamentos (métodos) às classes.
- Para ilustrar, vamos criar um método chamado *reset* que move um objeto Point para a origem:

In [1]:

```
1 class Point:
2     def reset(self):
3         self.x = 0
4         self.y = 0
5
6 p = Point()
7 p.reset()
8 print(p.x, p.y)
```

0 0

- Um método em python é formatado identicamente a uma função;
- A diferença de um método para uma função é que o método recebe um parâmetro chamado *self*;
- O parâmetro *self* é uma simples referência do próprio objeto (isto é, o objeto que está sendo invocado);

O que acontece se esquecermos o *self*?

In [2]:

```
1 class Point:
2     def reset():
3         pass
4
5 p = Point()
6 p.reset()
```

```
-----
-----
TypeError                                Traceback (most recent call
last)
<ipython-input-2-bec9ede0179d> in <module>()
      4
      5 p = Point()
----> 6 p.reset()
```

**TypeError:** reset() takes 0 positional arguments but 1 was given

- Python lança um erro não muito significativo explicando que você deveria ter passado um argumento para o método *reset*.

## Adicionando Métodos

- Vamos adicionar para nossa classe *Point* um método para mover o ponto para uma posição arbitrária e outro para calcular a distância entre dois pontos:

In [15]:

```
1 import math
2
3 class Point:
4     def move(self, x, y):
5         self.x = x
6         self.y = y
7
8     def reset(self):
9         self.move(0.0, 0.0)
10
11     def calculate_distance(self, p_2):
12         dist = math.sqrt((self.x - p_2.x)**2 +
13                           (self.y - p_2.y)**2)
14         return dist
```

- A classe acima tem 3 métodos;
  - O método move aceita dois argumentos (x e y) e os atribui aos atributos x e y do objeto self;
  - O método reset reaproveita a implementação de move, já que o reset é apenas um movimento para a origem;
  - O método calculate\_distance calcula a distância euclidiana entre dois pontos no plano;
- Para testarmos nossos objetos, vamos criar dois pontos e calcular a distância entre eles:

In [17]:

```
1 ponto1 = Point()
2 ponto2 = Point()
3
4 ponto1.reset()
5 ponto2.move(5,0)
6
7 print(ponto1.calculate_distance(ponto2))
8 assert(ponto1.calculate_distance(ponto2)==
9         ponto2.calculate_distance(ponto1))
```

5.0

Obs.: a assertiva acima é só um meio de validarmos que a distância de a para b é igual a distância de b para a;

- Agora vamos mover o ponto e calcular novamente a distância:

In [18]:

```
1 ponto1.move(3,4)
2 print(ponto1.calculate_distance(ponto2))
```

4.47213595499958

## Construtores

- Note que se não usarmos os métodos *move* ou *reset* da nossa classe *Point* ou ainda inicializarmos diretamente os valores de *x* e *y*, então teremos um objeto *Point* sem os atributos *x* e *y*;
  - Isso pode provocar erros, conforme segue:

In [8]:

```
1 ponto = Point()
2 ponto.x = 5
3 print(ponto.x)
4 print(ponto.y)
```

5

```
-----
-----
AttributeError                                Traceback (most recent call
last)
<ipython-input-8-c5a252d8e45e> in <module>()
      2 ponto.x = 5
      3 print(ponto.x)
----> 4 print(ponto.y)
```

**AttributeError:** 'Point' object has no attribute 'y'

- A mensagem de erro mostra que ocorreu um *AttributeError* na linha 5 porque o *y* não é um atributo da classe *Point*;
- Observe que o ideal é que todo novo objeto recebesse valores default ou que o usuário da classe *Point* fosse obrigado a atribuir valores para *x* e *y* ao instanciar um objeto dessa classe;
- A maioria das linguagens orientada a objetos possuem o conceito de construtor;
- Construtor é um método especial que cria e inicializa objetos quando eles são instanciados;
  - No python o método que inicializa objetos é o `__init__`
  - O underscore duplo significa que o python o interpreta como um método especial;

## Dica

Não use underscore duplo no início de identificadores de métodos pois caso o python adicione uma função com o mesmo nome da sua, então seu código vai quebrar;

- Para ilustrar o uso do `__init__`, vamos ajustar nossa classe *Point* para que o usuário dela seja obrigado a especificar as coordenadas quando ele instanciar o objeto *Point*:

In [22]:

```
1 ▼ class Point:
2 ▼     def __init__(self, x, y):
3         self.x = x
4         self.y = y
5
6 ▼     def move(self, x, y):
7         self.x = x
8         self.y = y
9
10 ▼    def reset(self):
11        self.move(0.0, 0.0)
12
13 ponto = Point(3, 5)
14 print(ponto.x, ponto.y)
```

3 5

- Agora se tentarmos criar um ponto sem passar as coordenadas, será lançado o erro **not enough arguments** (similar ao que recebemos antes no caso do *self*);
- Opcionalmente, podemos ainda passar **valores default** para os atributos x e y, para que esses valores sejam assumidos caso o usuário da classe não passe as coordenadas:

In [23]:

```
1 ▼ class Point:
2 ▼     def __init__(self, x = 0.0, y = 0.0):
3         self.x = x
4         self.y = y
5
6 ▼     def move(self, x, y):
7         self.x = x
8         self.y = y
9
10 ▼    def reset(self):
11        self.move(0.0, 0.0)
```

In [ ]:

```
1 p1 = Point()
```

In [27]:

```
1 p1 = Point()
2 print(p1.x, p1.y)
3 p2 = Point(3.0, 5.0)
4 print(p2.x, p2.y)
```

```
0.0 0.0
3.0 5.0
```

## Docstrings

- Apesar do python ser uma linguagem de fácil interpretação, precisamos documentar nossos códigos;
  - Principalmente quando estamos trabalhando em equipe ou fazendo projetos que podem ser escalados;
  - Além disso, em orientação a objetos é importante escrever a documentação das APIs da forma mais clara e concisa possível, explicando o que cada objeto, atributo e método faz;
- Através do uso de docstrings podemos adicionar a documentação dentro do próprio código;
  - Para tanto, colocamos a documentação em aspas simples ou dupla para documentações de linhas únicas ou as aspas simples ou duplas três vezes para textos que ficam em mais de uma linha;
  - A documentação deve seguir a indentação da classe, método, etc.
- Para ilustrar o uso do docstring, vamos documentar nossa classe Point:

In [28]:

```

1  import math
2
3  class Point:
4      "Representa coordenadas geométricas de um ponto no espaço bidimensional"
5
6      def __init__(self, x=0, y=0):
7          """Inicializa a posição de um novo ponto. x e y podem
8              ser especificados. Se eles não forem, as coordenadas
9              serão inicializadas na origem."""
10         self.move(x, y)
11
12     def move(self, x, y):
13         "Move um ponto para uma nova coordenada no espaço 2D."
14         self.x = x
15         self.y = y
16
17     def reset(self):
18         "Reposiciona um ponto na origem geométrica: (0, 0)"
19         self.move(0, 0)
20
21     def calculate_distance(self, other_point):
22         """Calcula a distância entre esse ponto e um segun-
23             do ponto passado como parâmetro. Depois, a distância
24             é retorna como um float."""
25         return math.sqrt(
26             (self.x - other_point.x) ** 2
27             + (self.y - other_point.y) ** 2
28         )

```

- Observe que ao usar a classe acima, o docstring fica disponível para o usuário da classe;

## Módulos

- Agora que sabemos como criar classes e instanciar objetos precisamos saber como organizá-los;
  - Para programas pequenos podemos colocar todas as classes em um único arquivo e apenas adicionar um script ao final do arquivo para fazer os objetos interagirem;
  - No entanto, quando o projeto começa a crescer pode ficar difícil achar classes que precisam ser modificadas entre tantas classes definidas em um único lugar;
  - Os módulos são simples arquivos python (\*.py) onde essas classes podem ser organizadas;
    - Ex.: dois arquivos python são dois módulos ;-)

- Se tivermos dois módulos no mesmo diretório podemos carregar classes, funções e métodos de um módulo para outro facilmente;

- Para ilustrar, vamos hipoteticamente implementar um sistema de ecommerce;
- Devemos armazenar muitos dados no database;
  - Assim sendo, podemos colocar todas as classes e funções relacionadas à base de dados dentro do módulo *database.py*;
- Assuma que existe uma classe chamada *Database* dentro do módulo *database.py*;
- Assuma ainda que existe um módulo chamado *products.py* responsável por fazer consultas relacionadas aos produtos;
- Existem algumas variações de sintaxe para importarmos a classe *Database* dentro do *products.py*:

In [ ]:

```
1 import database
2 db = database.Database()
3 # Do queries on db
```

- Na versão acima importamos o módulo *database* para o *namespace* do *products.py*
  - Namespace é uma lista de identificadores acessíveis para um módulo ou função;
- Assim, podemos acessar qualquer classe ou função de *database* usando a notação

**database.< something >**

- Alternativamente, podemos importar coisas específicas usando a notação *from...import*:

In [ ]:

```
1 from database import Database
2 db = Database()
3 # Do queries on db
```

- Se, por algum motivo, o módulo *products* já possuir uma classe chamada *Database* e não quisermos que haja conflito de nomes, então podemos colocar um apelido no *database.Database*:

In [ ]:

```
1 from database import Database as DB
2 db = DB()
3 # Do queries on db
```

- Podemos ainda importar vários itens em uma única instrução:

In [ ]:

```
1 from database import Database, Query #suponha que existe Query no módulo data
```

- Apesar de não recomendado, podemos também importar todas as classes e funções de um módulo usando o caractere *\**



In [ ]:

```
1 from database import *
```

- O importe de tudo não é recomendado, pois:
  - Prejudica a legibilidade do código;
  - Pode gerar problemas evitáveis no namespace (como conflito de identificadores), em função de objetos indesejáveis no namespace;

## Pacotes

- Conforme o número de módulos crescem é desejável acrescentar um novo nível de abstração para a abstração, uma forma de organizar módulos em uma hierarquia como de diretórios;
  - Podemos fazer isso através de pacotes que são diretórios que contém módulos;
  - Tudo que precisamos fazer para dizer que um diretório é um pacote é colocar dentro dele um arquivo (pode estar vazio) chamado `__init__.py`;
    - Se você esquecer de colocar esse arquivo, você não conseguirá importar módulos a partir desse diretório;
- Para ilustrar, vamos organizar nosso sistema de ecommerce em pacotes e módulos, conforme segue:

```
parent_directory/  
    main.py  
    ecommerce/  
        __init__.py  
        database.py  
        products.py  
        payments/  
            __init__.py  
            square.py  
            stripe.py
```

- Quando importamos módulos entre pacotes devemos ter algumas precauções;
- Existem basicamente duas formas de realizar esse importes: importes absolutos e importes relativos;

## Importes Absolutos

- Importes absolutos especificam o caminho completo do módulo, função ou classe;
  - Por exemplo, se você quiser acessar à classe `Products` dentro do módulo `products.py`, a sintaxe do importe absoluto é a seguinte:

In [ ]:

```
1 import ecommerce.products
2 product = ecommerce.products.Product()
```

Ou

In [ ]:

```
1 from ecommerce.products import Product
2 product = Product()
```

Ou ainda

In [ ]:

```
1 from ecommerce import products
2 product = products.Product()
```

- Existe ainda uma outra opção que é exportar seus módulos para o PYTHONPATH.
- Dessa forma eles poderão ser importados por qualquer módulo em qualquer lugar.
- Para exportar scripts para o PYTHONPATH no linux basta abrir o prompt e escrever o seguinte comando:

In [ ]:

```
1 export PYTHONPATH=PATH_OF_YOUR_MODULES
```

- Onde *PATH\_OF\_YOUR\_MODULES* é o caminho para os modelos que você deseja exportar;
- Perceba que este comando só funcionará para a sessão do terminal onde você o adicionou;
  - para que a biblioteca seja acessível para outras sessões de terminal, exporte o PYTHONPATH no *bashrc*, da seguinte forma:
    1. Abra o arquivo *~/.bashrc* no seu editor favorito (ex.: *gedit ~/.bashrc*)
    2. No final do arquivo *~/.bashrc* adicione o comando
      - `export PYTHONPATH=PATH_OF_YOUR_MODULES`
    3. Salve o arquivo;
- Para mais informações, acesso o [tutorial \(https://bic-berkeley.github.io/psych-214-fall-2016/using\\_pythonpath.html\)](https://bic-berkeley.github.io/psych-214-fall-2016/using_pythonpath.html);

## Importes Relativos

- As importações relativas são basicamente uma forma de dizer encontrar uma classe, uma função ou um módulo à medida que ele é posicionado em relação ao módulo atual;
  - Por exemplo, se você estiver trabalhando no módulo *products* e quiser importar a classe *Database* a partir do módulo *database*, você pode fazer:

In [ ]:

```
1 from .database import Database
```

- O ponto na frente de database informa ao python que o módulo database está no mesmo pacote que o módulo corrente;
  - Nesse caso o pacote atual é o ecommerce.
- Por outro lado, se você estiver editando um módulo chamado paypal que fica no pacote ecommerce.payments, nós poderíamos querer usar a classe Database também;
- Para acessar ela na classe pai, basta usar dois pontos seguidos, conforme segue:

In [ ]:

```
1 from ..database import Database
```

- Finalmente, podemos importar códigos diretamente de pacotes ao invés de módulos dentro de pacotes.
- Por exemplo se quisermos importar a classe Database diretamente do pacote ecommerce, temos que adicionar a seguinte linha no arquivo `__init__.py` do pacote ecommerce:

In [ ]:

```
1 from .database import Database
```

- Com isso, podemos importar Database, por exemplo, a partir do arquivo main.py da seguinte maneira:

In [ ]:

```
1 from ecommerce import Database
```

- Perceba que a maneira tradicional (`from ecommerce.database import Database`) ainda funcionará normalmente;

## Dica

- Às vezes ajuda pensar no `__init__.py` como se fosse um arquivo *ecommerce.py* (se ele fosse um módulo ao invés de um pacote);
- O `__init__.py` funcionará como um ponto de contato entre outros módulos, mas o código pode ser internamente organizado em diferentes módulos ou até subpacotes;

## Exemplo prático da biblioteca sklearn

- O sklearn é uma biblioteca feita em python com algoritmos de aprendizagem de máquina;
  - Acesse o [github do sklearn](https://github.com/scikit-learn/scikit-learn/tree/master/sklearn) (<https://github.com/scikit-learn/scikit-learn/tree/master/sklearn>);
- Dentro do pacote sklearn existe um subpacote chamado [ensemble](https://github.com/scikit-learn/scikit-learn/tree/master/sklearn/ensemble) (<https://github.com/scikit-learn/scikit-learn/tree/master/sklearn/ensemble>);
  - Observe que nele existe um módulo chamado [forest.py](https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/ensemble/forest.py) (<https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/ensemble/forest.py>);
  - *RandomForestClassifier* é uma classe que fica dentro do módulo forest.py;
  - Observe que podemos importar a classe *RandomForestClassifier* com o seguinte comando:

In [1]:

```
1 from sklearn.ensemble import RandomForestClassifier
```

- Mas como importamos o RandomForestClassifier diretamente do pacote ensemble se a classe fica dentro do módulo *forest.py*?
- Veja que o import abaixo também funciona:

In [3]:

```
1 from sklearn.ensemble.forest import RandomForestClassifier
```

- A resposta a pergunta anterior pode ser vista no `__init__.py` ([https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/ensemble/\\_\\_init\\_\\_.py](https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/ensemble/__init__.py)) da pacote ensemble.
  - Dentro do `__init__.py` existe a seguinte linha:
    - `from .forest import RandomForestClassifier`

## Organizando o conteúdo dos módulos

- Dentro de qualquer módulo é possível especificar variáveis, classes ou funções;
- Isso pode ser maneira esperta de guardar o estado global de objetos sem conflitos de namespace;
  - Por exemplo, nós importamos e instanciamos a classe *Database* em vários módulos;
  - Entretanto, faz mais sentido ter um objeto da classe *Database* global que seja acessível através do módulo *database*;
  - O módulo *database* seria implementado assim:

In [ ]:

```
1 class Database:
2     # the database implementation
3     pass
4
5 db = Database() #objeto a ser compartilhado
```

- Assim, podemos importar o objeto *db* da seguinte forma:

In [ ]:

```
1 from ecommerce.database import db
```

- O problema dessa abordagem é que o objeto *database* é criado imediatamente quando ele for importado a primeira vez, o que ocorre normalmente durante a inicialização da aplicação;
  - Isso pode tornar o processo de inicialização da aplicação lento;
- Uma alternativa para contornar esse problema é usar um método que cria o objeto *db* somente quando for necessário:

In [ ]:

```
1 ▾ class Database:
2     # the database implementation
3     pass
4
5     db = None
6
7 ▾ def initialize_database():
8     global db
9     db = Database()
```

- O palavra chave **global** explica ao python que a variável db da linha 7 é a mesma da linha 5 no script acima;
  - Se não fizéssemos isso, a variável db teria um escopo local, isto é, ela seria destruída assim que a função *initialize\_database* terminasse;
- Entretanto, esse código ainda é perigoso, porque a função *initialize\_database* poderia ser chamada mais de uma vez, reiniciando o estado do objeto db;
- Para minimizar esse problema, colocamos nosso *startup code* em uma função (convencionalmente chamada de main) e só a executamos quando sabemos que estamos executando-a como script, mas não quando o código está sendo importado.
  - Segue abaixo como faríamos isso:

In [ ]:

```
1 ▾ class UsefulClass:
2     '''Essa classe pode ser útil para outros módulos.'''
3     pass
4
5 ▾ def main():
6     '''cria uma classe útil e faz algo com ela para o nosso
7     módulo.'''
8
9     useful = UsefulClass()
10    print(useful)
11
12 ▾ if __name__ == "__main__":
13     main()
```

- A linha 12 do código acima também é útil quando queremos testar um código no próprio módulo;
  - Isso porque todos módulos tem um `__name__` especial que é especificado quando ele é importado.
  - Quando o módulo é executado diretamente com *python module.py* e ele nunca foi importado, então `__name__` assume a string `__main__`;

## Classes internas

- Classes podem ser definidas em qualquer lugar;
  - Normalmente, elas são definidas a nível de módulo, mas elas podem ser definidas também dentro de funções e métodos.
  - Veja abaixo um exemplo:

In [4]:

```
1  def format_string(string, formatter=None):
2      '''Formata uma string usando um objeto foratter, que
3      deve possuir um método format() que recebe como parâ-
4      metro uma string.'''
5
6      class DefaultFormatter:
7          '''Formata a string em title case (deixando a pri-
8          meira letra de cada palavra maiúscula).'''
9          def format(self, string):
10             return str(string).title()
11
12     if not formatter:
13         formatter = DefaultFormatter()
14
15     return formatter.format(string)
16
17 hello_string = "olá pessoal, o que vocês estão achando da disciplina?"
18 print(" input: " + hello_string)
19 print("output: " + format_string(hello_string))
```

input: olá pessoal, o que vocês estão achando da disciplina?  
output: Olá Pessoal, O Que Vocês Estão Achando Da Disciplina?

- A função *format\_string* recebe uma string e um objeto que formata essa string, o qual é opcional;
  - Se um objeto formatador não for fornecido, então a classe *DefaultFormatter* é usada;

## Quem pode acessar os dados

- A maioria das linguagens orientadas a objetos possuem o conceito de controle de acesso;
  - Nessas linguagens atributos e métodos podem ser privados, protegidos ou públicos;
- No python não existe isso;
  - Python não acredita em leis que te forcem a algo que pode ser prejudicial no futuro;
  - O python fornece diretrizes (não obrigatórias) e boas práticas;
  - Assim sendo, tecnicamente todos os atributos e métodos são públicos.
    - Se você quiser que um método seja privado, devemos sugerir isso no docstring do método;
    - Por convenção podemos ainda colocar um underscore na frente de um atributo ou método;
      - Programadores python vão interpretar isso como um sinal de que aquele atributo/método é privado;
    - Outra possibilidade é colocar um underscore duplo na frente do identificador do atributo ou método;
      - Ao colocar underscore duplo, o python realiza name mangling;
      - Veja um exemplo:

In [8]:

```
1 ▼ class SecretString:
2     '''Uma maneira nada segura de armazenar uma string
3     que contém um segredo.'''
4 ▼     def __init__(self, plain_string, pass_phrase):
5         self.__plain_string = plain_string
6         self.__pass_phrase = pass_phrase
7 ▼     def decrypt(self, pass_phrase):
8         '''Só mostra o segredo se o senha estiver certa.'''
9 ▼         if pass_phrase == self.__pass_phrase:
10             return self.__plain_string
11 ▼         else:
12             return ''
```

In [6]:

```
1 secret_string = SecretString("ACME: Top Secret", "antwerp")
```

In [9]:

```
1 print(secret_string.decrypt("antwerp"))
```

ACME: Top Secret

- Se tentarmos:

In [10]:

```
1 print(secret_string.__plain_text)
```

```
-----
-----
AttributeError                                Traceback (most recent call
last)
<ipython-input-10-376091f5ceea> in <module>()
----> 1 print(secret_string.__plain_text)
```

AttributeError: 'SecretString' object has no attribute '\_\_plain\_text'

- Entretanto, podemos facilmente acessar a senha e o segredo:

In [13]:

```
1 print(secret_string._SecretString__pass_phrase)
2 print(secret_string._SecretString__plain_string)
```

antwerp

ACME: Top Secret

- O nome magling do python coloca o nome da classe como prefixo quando usamos o underscore duplo;
- Em geral, programadores python não irão mexer em variáveis como underscore duplo ou mesmo simples;
  - Salvo se eles tiverem uma boa razão para fazer isso;

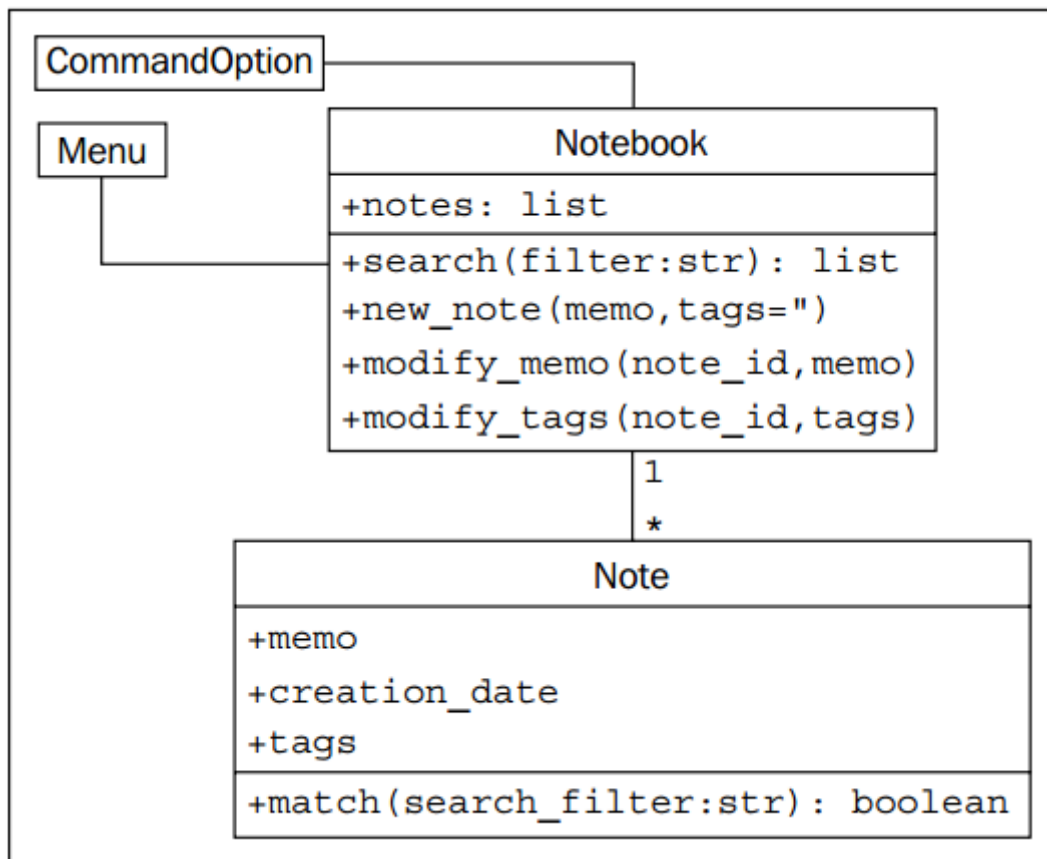
# Bibliotecas de Terceiros

- O Python vem com uma adorável biblioteca padrão, que é uma coleção de pacotes e módulos que estão disponíveis em todas as máquinas que executam o Python.
- Entretanto, às vezes precisamos de bibliotecas de terceiros;
  - Para procurar bibliotecas de terceiros use o [Python Package Index \(PyPi\)](http://pypi.python.org/) (<http://pypi.python.org/>).
  - Uma vez que você identificou a biblioteca que você quer usar, basta usar o *pip* para instalá-la;

## Estudo de Caso

- Vamos agora tentar aplicar os conceitos aprendidos durante essa aula;
- Desenvolveremos uma aplicação para gerenciar anotações;
- Requisitos no nosso gerenciador de anotações:
  - Em cada anotação devemos guardar a data que ela foi feita e tags para que elas sejam facilmente consultadas;
  - Deve ser possível alterar notas;
  - Deve ser possível buscar notas;
  - Tudo isso deve ser feito via linhas de comando no terminal;
- Necessitaremos de um objeto chamado *Note* e de um *Notebook* que será um container de Notes;
- Para armazenar as datas das notas iremos usar um biblioteca built-in do python;
- Tags serão strings;
- A classe *Note* terá os seguintes atributos:
  - *id* (um identificador único);
  - *memo* (a anotação gravada);
  - *tags*;
  - *creation\_data*;
- Para facilitar o processo de busca de uma anotação, iremos fornecer o método *match* que receberá uma string e verificará se a anotação corresponde a ela;
- A classe *Notebook* terá uma lista de Notes como atributo;
- *Notebook* terá os métodos:
  - *search*: que retorna uma lista de anotações filtradas;
  - *add*: para adicionar uma nova anotação;
  - *modify\_memo*: para modificar uma anotação através do *id* dela;
  - *modify\_tags*: para modificar tags através do *id*;
- Além disso, é desejável que possamos fazer um menu (para que depois possamos fazer uma GUI) e uma opção da aplicação por linha de comando;
- Vejamos como fica o diagrama de classes do nosso embrionário design orientada a objetos de um sistema de gerenciamentos de notas:





- Antes de começarmos a codificar, vamos projetar a estrutura de nosso projeto:
  - O menu deve ter seu próprio módulo, uma vez que ele será um script executável;
  - As classes Notebook e Note podem ficar juntas em um módulo chamado *notebook*;
  - Criaremos ainda um módulo vazio chamado *command\_option* para que possamos lembrar no futuro de implementarmos essa opção;

```

parent_directory/
    notebook.py
    menu.py
    command_option.py
  
```

- Em termos de codificação, vamos começar criando a classe *Note* (dentro do arquivo *notebook.py*), uma vez que ela parece ser a mais simples;
- O conteúdo será o seguinte (exclua a linha com o magic command `%% file notebook.py`):

In [8]:

```
1  ▾ %%writefile notebook.py
2  import datetime
3
4  # A variável seguinte é utilizada para guardar o prox. id disponível para uma
5  last_id = 0
6
7  ▾ class Note:
8      '''Representa uma nota em um notebook. Pode-se combiná-la com
9      uma string e armazenar tags para cada nota.'''
10
11  ▾ def __init__(self, memo, tags=''):
12      '''inicializa uma nota com uma anotação (string) e uma
13      tag opcional. A data de criação e o id são automatica-
14      mente definidos para cada nota.'''
15      self.memo = memo
16      self.tags = tags
17      self.creation_data = datetime.date.today()
18      global last_id
19      last_id += 1
20      self.id = last_id
21
22  ▾ def match(self, term):
23      '''Determina se essa nota corresponde com o string term
24      passada como parâmetro. É retornado True se houver cor-
25      respondência e falso, caso contrário.
26
27      A busca é case sensitive e faz correspondência tanto no
28      texto quando nas tags
29      '''
30      return term in self.memo or term in self.tags
```

Writing notebook.py

- Antes de continuarmos nosso projeto, vamos testar nosso código;
- Observe que é importante testarmos frequentemente nossos códigos porque as coisas podem (normalmente...rsrs) não funcionar conforme planejamos.

In [1]:

```
1  from notebook import Note
2
3  n1 = Note('Minha primeira anotação')
4  n2 = Note('Uma nova anotação')
```

In [2]:

```
1  n1.id
```

Out[2]:

1

In [3]:

```
1 n2.id
```

Out[3]:

2

In [4]:

```
1 n1.match('primeira')
```

Out[4]:

True

In [5]:

```
1 n2.match('segunda')
```

Out[5]:

False

- Como podemos ver, tudo está funcionando conforme o esperado.
- Agora vamos criar a classe Notebook dentro do módulo *notebook.py*:
  - Obs.: Ao copiar e colar o código dentro do módulo *notebook.py*, não esqueça de remover a primeira linha com o conteúdo (`%%writefile -a notebook.py`)

In [9]:

```
1 ▾ %%writefile -a notebook.py
2
3
4 ▾ class Notebook:
5     '''Representa uma coleção de notas que podem possuir,
6     tags associadas, modificadas e buscadas.'''
7
8     def __init__(self):
9         '''Inicializa um notebook com uma lista vazia de anotações.'''
10        self.notes = []
11
12     def new_note(self, memo, tags=''):
13         '''Cria uma nova nota e a adiciona a lista.'''
14        self.notes.append(Note(memo, tags))
15
16     def modify_memo(self, note_id, memo):
17         '''Encontra a anotação pelo id e modifica o texto com o
18         novo memo passado como parâmetro.'''
19        for note in self.notes:
20            if note.id == note_id:
21                note.memo = memo
22                break
23
24     def modify_tags(self, note_id, tags):
25         '''Encontra a anotação pelo id e modifica as tags com as
26         novas tags passadas como parâmetro.'''
27        for note in self.notes:
28            if note.id == note_id:
29                note.tags = tags
30                break
31
32     def search(self, term):
33         '''Procura por todas as notas que possuem correspondência
34         com a string term passada como parâmetro.'''
35        return [note for note in self.notes if note.match(term)]
```

Appending to notebook.py

- Antes de mais nada, vamos testar nossa classe Notebook:

In [1]:

```
1  from notebook import Note, Notebook
```

- Primeiramente, vamos criar um objeto da classe Notebook e adicionar duas notas:

In [4]:

```
1 n = Notebook()
2 n.new_note('hello world')
3 n.new_note('hello again')
4 n.notes
```

Out[4]:

```
[<notebook.Note at 0x7fad64e7f550>, <notebook.Note at 0x7fad64e7f128>]
```

- Vejamos abaixo os ids e anotações nas duas notas:

In [5]:

```
1 print(n.notes[0].id, n.notes[0].memo)
2 print(n.notes[1].id, n.notes[1].memo)
```

```
1 hello world
2 hello again
```

- Se pesquisarmos pelo termo "hello", serão recuperados duas notas, uma vez que ambas as notas possuem esse termo:

In [6]:

```
1 n.search('hello')
```

Out[6]:

```
[<notebook.Note at 0x7fad64e7f550>, <notebook.Note at 0x7fad64e7f128>]
```

- Por outro lado, se pesquisarmos por 'world', recuperaremos apenas um objeto Note;

In [7]:

```
1 n.search('world')
```

Out[7]:

```
[<notebook.Note at 0x7fad64e7f550>]
```

- Abaixo vamos modificar o objeto cujo id é 1:

In [8]:

```
1 n.modify_memo(1, 'hi world')
```

In [9]:

```
1 n.notes[0].memo
```

Out[9]:

```
'hi world'
```

- Nossa classe Notebook está funcionando bem;
- Entretanto, observem que o método `modify_memo` e `modify_tags` fazem praticamente a mesma coisa;
  - Ter códigos repetidos assim não é uma boa prática;
  - Para melhorar isso, vamos criar um novo método chamado **`find_note`**, que retornará um note dado um id;
  - Os métodos `modify_memo` e `modify_tags` usarão esse método para realizar a busca do objeto Note e modificarão os campos correspondentes (memo e tags);
  - Note que `find_note` deve ser um método de uso interno e, portanto, usaremos underscore duplo na frente de seu identificador.
- A classe Notebook ficará conforme segue:

In [ ]:

```
1  class Notebook:
2      '''Representa uma coleção de notas que podem possuir,
3      tags associadas, modificadas e buscadas.'''
4
5  def __init__(self):
6      '''Inicializa um notebook com uma lista vazia de anotações.'''
7      self.notes = []
8
9  def new_note(self, memo, tags=''):
10     '''Cria uma nova nota e a adiciona a lista.'''
11     self.notes.append(Note(memo, tags))
12
13  def __find_note(self, note_id):
14     '''Encontra um Note dado um id. Caso não encontre,
15     retorna um objeto nulo (None)'''
16     for note in self.notes:
17         if note.id == note_id:
18             return note
19     return None
20
21  def modify_memo(self, note_id, memo):
22     '''Encontra a anotação pelo id e modifica o texto com o
23     novo memo passado como parâmetro.'''
24     note = self.__find_note(note_id)
25     if note != None:
26         note.memo = memo
27
28  def modify_tags(self, note_id, tags):
29     '''Encontra a anotação pelo id e modifica as tags com as
30     novas tags passadas como parâmetro.'''
31     note = self.__find_note(note_id)
32     if note != None:
33         note.tags = tags
34
35  def search(self, term):
36     '''Procura por todas as notas que possuem correspondência
37     com a string term passada como parâmetro.'''
38     return [note for note in self.notes if note.match(term)]
```

- Agora vamos criar o módulo *menu.py*:

In [14]:

```
1  import sys
2
3  from notebook import Notebook, Note
4
5  class Menu:
6      '''Mostra um menu e aciona as ações apropriadas com base
7      nas opções escolhidas.'''
8      def __init__(self):
9          self.notebook = Notebook()
10         self.choices = {
11             "1": self.show_notes,
12             "2": self.search_notes,
13             "3": self.add_note,
14             "4": self.modify_note,
15             "5": self.quit
16         }
17     def display_menu(self):
18         print("""
19         Notebook Menu
20
21         1. Mostrar todas as Notas
22         2. Buscar Notas
23         3. Adicionar Nota
24         4. Modificar Nota
25         5. Sair
26         """)
27     def run(self):
28         '''Mostra o menu e aciona a opção escolhida.'''
29         while True:
30             self.display_menu()
31             choice = input("Escolha uma opção: ")
32             action = self.choices.get(choice)
33             if action:
34                 action()
35             else:
36                 print("{0} não é uma opção válida".format(choice))
37
38     def show_notes(self, notes=None):
39         if not notes:
40             notes = self.notebook.notes
41         for note in notes:
42             print("{0}: {1}\n{2}".format(note.id, note.tags, note.memo))
43
44     def search_notes(self):
45         term = input("Buscar por: ")
46         notes = self.notebook.search(term)
47         self.show_notes(notes)
48
49     def add_note(self):
50         memo = input("Entre com a anotação: ")
51         self.notebook.new_note(memo)
52         print("Sua anotação foi adicionada.")
53
54     def modify_note(self):
55         id = input("Entre com o id da anotação: ")
56         memo = input("Entre com a anotação: ")
57         tags = input("Entre com as tags: ")
58         if memo:
59             self.notebook.modify_memo(int(id), memo)
```



```

60     if tags:
61         self.notebook.modify_tags(int(id), tags)
62
63     def quit(self):
64         print("Obrigado por usar nosso sistema!")
65         sys.exit(0)
66
67     if __name__ == "__main__":
68         Menu().run()

```

Notebook Menu

1. Mostrar todas as Notas
2. Buscar Notas
3. Adicionar Nota
4. Modificar Nota
5. Sair

Escolha uma opção: 3

Entre com a anotação: Aula de P00 - Objetos interagem através de trocas de mensagens. Objetos são instâncias de classes.  
Sua anotação foi adicionada.

Notebook Menu

1. Mostrar todas as Notas
2. Buscar Notas
3. Adicionar Nota
4. Modificar Nota
5. Sair

Escolha uma opção: 3

Entre com a anotação: Objetos possuem um estado que é definido com base nos dados dos atributos.  
Sua anotação foi adicionada.

Notebook Menu

1. Mostrar todas as Notas
2. Buscar Notas
3. Adicionar Nota
4. Modificar Nota
5. Sair

Escolha uma opção: 1

6:  
Aula de P00 - Objetos interagem através de trocas de mensagens. Objetos são instâncias de classes.

7:  
Objetos possuem um estado que é definido com base nos dados dos atributos.

Notebook Menu

1. Mostrar todas as Notas
2. Buscar Notas
3. Adicionar Nota
4. Modificar Nota
5. Sair

```
Escolha uma opção: 2
Buscar por: 1
6:
Aula de P00 - Objetos interagem através de trocas de mensagens. Objeto
s são instâncias de classes.
7:
Objetos possuem um estado que é definido com base nos dados dos atributos.
```

#### Notebook Menu

1. Mostrar todas as Notas
2. Buscar Notas
3. Adicionar Nota
4. Modificar Nota
5. Sair

```
Escolha uma opção: 5
Obrigado por usar nosso sistema!
```

```
An exception has occurred, use %tb to see the full traceback.
```

```
SystemExit: 0
```

```
/home/dwan/anaconda3/lib/python3.6/site-packages/IPython/core/interactiveshell.py:3334: UserWarning: To exit: use 'exit', 'quit', or Ctrl-D.
warn("To exit: use 'exit', 'quit', or Ctrl-D.", stacklevel=1)
```

- No menu acima, criamos um dicionário onde as chaves são as opções e os valores são os métodos correspondentes às opções.
- Outra coisa que vale ressaltar é que quando recebemos valores da função `input()`, recebemos como string;
  - Como o `id` é um `int`, então ele foi passado como parâmetro como `int` para o método modificador.

## Exercício Avaliativo

- Faça o download da apostila da caelum **py14**. Para realizar o download gratuito basta fornecer o seu e-mail.
- Com a apostila em mãos, faça os exercícios da seção 7.13. 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;