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 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]:

```
class MyFirstClass:
    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/);

- 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]:

```
a = MyFirstClass()
b = MyFirstClass()

print(a)
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]:

```
class Point:
    pass

p1 = Point()
p2 = Point()
```

In [4]:

```
p1.x = 5
p1.y = 4

p2.x = 3
p2.y = 6
```

In [5]:

```
print(p1.x, p1.y)
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]:

```
class Point:
    def reset(self):
        self.x = 0
        self.y = 0

p = Point()
p.reset()
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 esquercemos o self?

In [2]:

```
class Point:
    def reset():
        pass

p = Point()
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]:

```
import math

class Point:
    def move(self, x, y):
        self.x = x
        self.y = y

    def reset(self):
        self.move(0.0, 0.0)

    def calculate_distance(self, p_2):
        dist = math.sqrt((self.x - p_2.x)**2 +
                         (self.y - p_2.y)**2)
        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]:

```
ponto1 = Point()
ponto2 = Point()

ponto1.reset()
ponto2.move(5,0)

print(ponto1.calculate_distance(ponto2))
assert(ponto1.calculate_distance(ponto2)==
      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]:

```
ponto1.move(3,4)
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 undescore duplo significa que o python o interpreta como um método especial;

Dica

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

- Para ilastrur o uso do __**init**__, vamos ajustar nossa classe Point para que o usário dela seja obrigado a especificar as coordenadas quando ele instanciar o objeto Point:

In [22]:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def move(self, x, y):
        self.x = x
        self.y = y

    def reset(self):
        self.move(0.0, 0.0)

ponto = Point(3, 5)
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*);

- Opcionamente, 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]:

```
class Point:
    def __init__(self, x = 0.0, y = 0.0):
        self.x = x
        self.y = y

    def move(self, x, y):
        self.x = x
        self.y = y

    def reset(self):
        self.move(0.0, 0.0)
```

In [ ]:

```
p1 = Point()
```

In [27]:

```
p1 = Point()
print(p1.x, p1.y)
p2 = Point(3.0, 5.0)
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 identação da classe, método, etc.
- Para ilustrar o uso do docstring, vamos documentar nossa classe Point:

In [28]:

```python
import math

class Point:
    "Representa coordenadas geométricas de um ponto no espaço bidimensional"

    def __init__(self, x=0, y=0):
        """Inicializa a posição de um novo ponto. x e y podem
           ser especificados. Se eles não forem, as coordenadas
           serão inicializadas na origem."""
        self.move(x, y)

    def move(self, x, y):
        "Move um ponto para uma nova coordenada no espaço 2D."
        self.x = x
        self.y = y

    def reset(self):
        "Reposiciona um ponto na origem geométrica: (0, 0)"
        self.move(0, 0)

    def calculate_distance(self, other_point):
        """Calcula a distância entre esse ponto e um segun-
        do ponto passado como parâmetro. Depois, a distância
        é retorna como um float."""
        return math.sqrt(
            (self.x - other_point.x) ** 2
            + (self.y - other_point.y) ** 2
        )
```

- 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 interegirem;
    - 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 eccomerce;
- 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* reponsável por fazer consultas relacionadas aos produtos;
- Existem algumas variações de sintaxe para importarmos a classe Database dentro do *products.py*:

In [ ]:

```
import database
db = database.Database()
# 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 [ ]:

```
from database import Database
db = Database()
# 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 [ ]:

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

- Podemos ainda importar vários itens em uma única instrução:

In [ ]:

```
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 [ ]:

```
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 aquivo (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 eccomerce 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 [ ]:

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

Ou ainda

In [ ]:

```
from ecommerce import products
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 [ ]:

```
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 [ ]:

```
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 eccomerce.

- Por outro lado, se você estiver editando um módulo chamado paypal que fica no pacote ecommerce.payments, nós poderíamos quere usar a classe Database também;
- Para acessar ela no 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 eccomerce, temos que adicionar a seguinte linha no arquivo **__init__.py** do pacote eccomerce:

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 um 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);
- Dentro do pacote sklearn existe um subpacote chamado 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);
  - *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) 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 variávies, 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]:

```python
def format_string(string, formatter=None):
    '''Formata uma string usando um objeto foratter, que
    deve possuir um método format() que recebe como parâ-
    metro uma string.'''

    class DefaultFormatter:
        '''Formata a string em title case (deixando a pri-
        meira letra de cada palavra maiúscula).'''
        def format(self, string):
            return str(string).title()
    if not formatter:
        formatter = DefaultFormatter()

    return formatter.format(string)

hello_string = "olá pessoal, o que vocês estão achando da disciplina?"
print(" input: " + hello_string)
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 forçam 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 exempĺo:

In [8]:

```
class SecretString:
    '''Uma maneira nada segura de armazenar uma string
    que contém um segredo.'''
    def __init__(self, plain_string, pass_phrase):
        self.__plain_string = plain_string
        self.__pass_phrase = pass_phrase
    def decrypt(self, pass_phrase):
        '''Só mostra o segredo se o senha estiver certa.'''
        if pass_phrase == self.__pass_phrase:
            return self.__plain_string
        else:
            return ''
```

In [6]:

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

In [9]:

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

```
ACME: Top Secret
```

- Se tentarmos:

In [10]:

```
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 facialmente acessar a senha e o segredo:

In [13]:

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

```
antwerp
ACME: Top Secret
```

- O name 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/)
  - Uma vez que você identificou a biblioteca que você quer usar, basta usar o *pip* para instalá-la;

In [ ]:
```
1
```

In [ ]:
```
1
```

In [ ]:
```
1
```

In [ ]:
```
1
```

In [ ]:
```
1
```

In [ ]:
```
1
```

In [ ]:
```
1
```

In [ ]:
```
1
```

In [ ]:
```
1
```

In [ ]:
```
1
```

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 mãoes, faça os execícios 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;