

PYTHON 3

*UMA ABORDAGEM PRÁTICA SOBRE A LINGUAGEM DE
PROGRAMAÇÃO*

Graduando Júlio César Machado Álvares

Prof. Dr. Laerte Mateus Rodrigues





Júlio César Machado Álvares

*Graduando em Engenharia de Computação pelo
Instituto Federal de Minas gerais - campus Bambuí*

Utilizo python a quase 2 anos.

Lattes: <http://lattes.cnpq.br/4037573872440424>



github.com/juliocmalvares

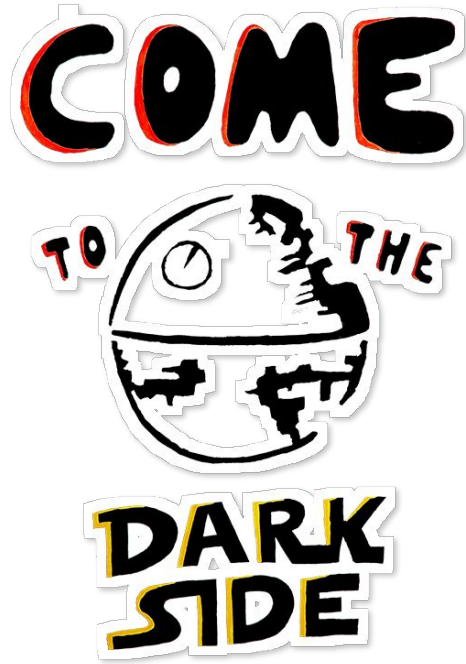
E-mail: juliocmalvares07@gmail.com

SUMÁRIO

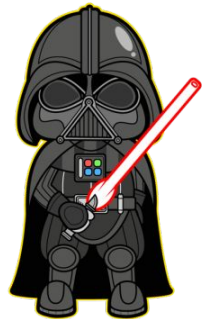
1. História e diferenças
2. Introdução
3. Variáveis
4. Operações matemáticas e lógicas
5. Listas
6. Tuplas
7. Condições e Loops
8. List Comprehensions
9. Strings
10. Dicionários
11. Funções, Métodos e Procedimentos
12. Manipulação de Arquivos
13. Módulos
14. Funções Interessantes



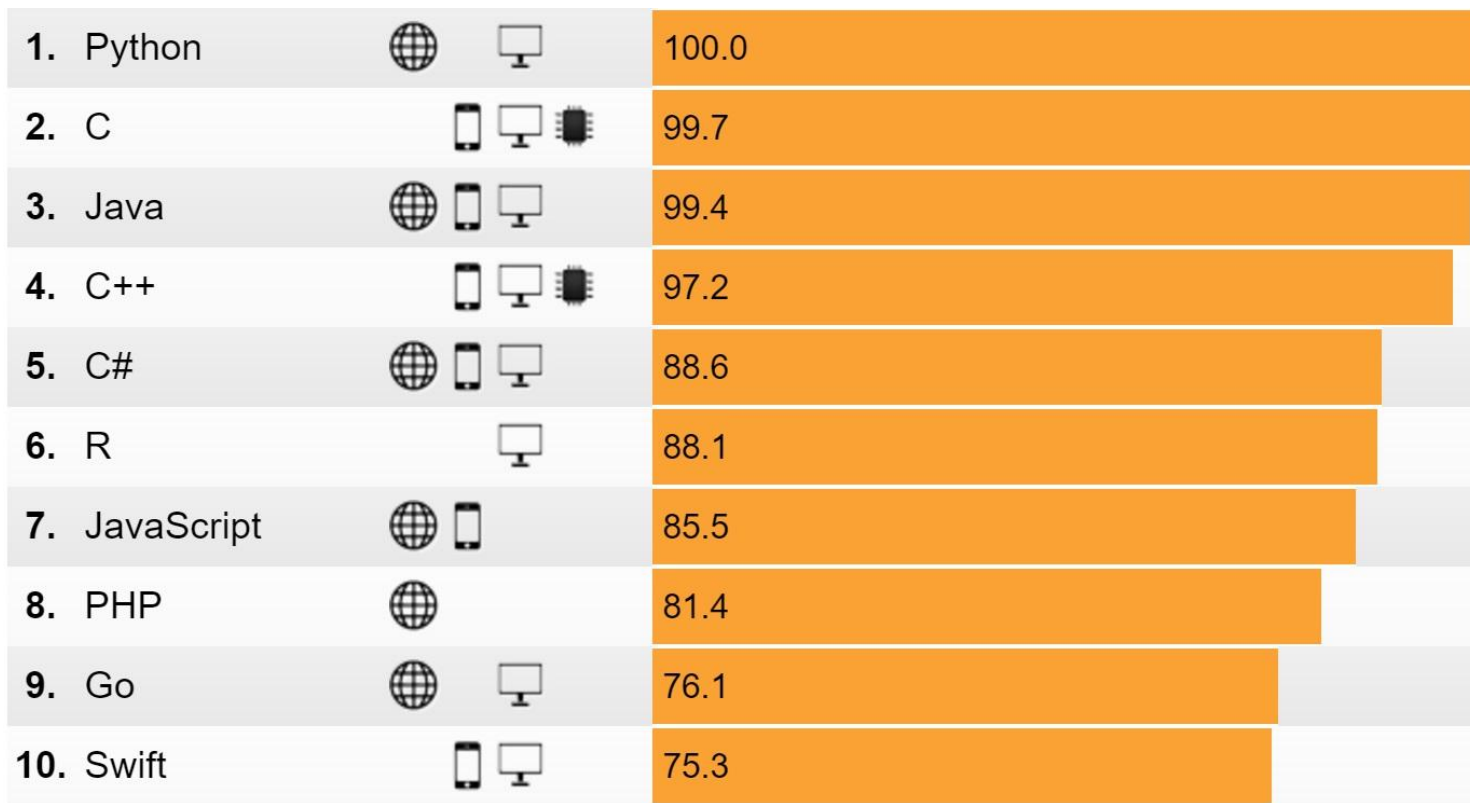
O QUE É PYTHON ??



PYTHON É O LADO NEGRO DA PROGRAMAÇÃO, AMIGOS
PADAWANS !



POR QUE APRENDER PYTHON?



Fonte: (<https://www.extremetech.com/computing/252987-python-tops-list-2017s-popular-programming-languages>)

POR QUE APRENDER PYTHON?

May 2018	May 2017	Change	Programming Language	Ratings	Change
1	1		Java	16.380%	+1.74%
2	2		C	14.000%	+7.00%
3	3		C++	7.668%	+2.92%
4	4		Python	5.192%	+1.64%
5	5		C#	4.402%	+0.95%
6	6		Visual Basic .NET	4.124%	+0.73%
7	9	▲	PHP	3.321%	+0.63%
8	7	▼	JavaScript	2.923%	-0.15%
9	-	▲▲	SQL	1.987%	+1.99%
10	11	▲	Ruby	1.182%	-1.25%
11	14	▲	R	1.180%	-1.01%
12	18	▲▲	Delphi/Object Pascal	1.012%	-1.03%
13	8	▼▼	Assembly language	0.998%	-1.86%
14	16	▲	Go	0.970%	-1.11%
15	15		Objective-C	0.939%	-1.16%
16	17	▲	MATLAB	0.929%	-1.13%
17	12	▼▼	Visual Basic	0.915%	-1.43%
18	10	▼▼	Perl	0.909%	-1.69%
19	13	▼▼	Swift	0.907%	-1.37%
20	31	▲▲	Scala	0.900%	+0.18%

Fonte: (<https://www.tiobe.com/tiobe-index/>)

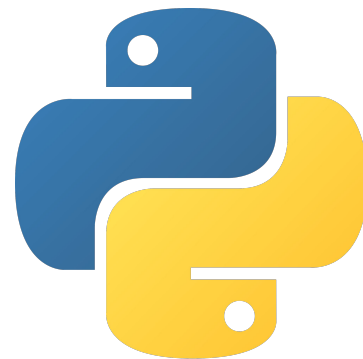


É HORA DA HISTÓRIA !

O Python é uma linguagem de alto nível, interpretada, de script, imperativa, orientada a objetos, funcional, de tipagem dinâmica e forte.

Criada por Guido van Rossum em 1989.

Atualmente é desenvolvida de forma comunitária e gerenciada pela *Python Software Foundation*. ([Python.org](https://python.org))



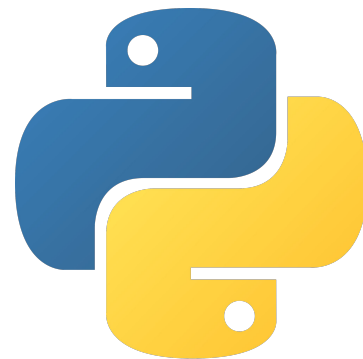
COMPILAÇÃO VS. INTERPRETAÇÃO

Compilação:

1. Código fonte convertido para a arquitetura;
2. Tempo de execução rápido;
3. Otimização de código-fonte.

Interpretação:

1. Código fonte executado pelo programa que o lê;
2. Tempo de execução lento;
3. Não apresenta otimização de código-fonte.



1. Scripts

Programa escrito para um sistema de tempo de execução

→ **Utilidade**

Automatiza execução de tarefas.

→ **Flexibilidade**

Permite que pequenos scripts sejam combinados em programas complexos

→ **Variabilidade**

Praticamente qualquer tarefa pode ser automatizada em quase todos os ambientes. (Páginas web, OS, Sistemas Embarcados, Games, etc)

VERSÃO A SER UTILIZADA:

PYTHON 3.5.2

INSTALAÇÃO

GNU/Linux

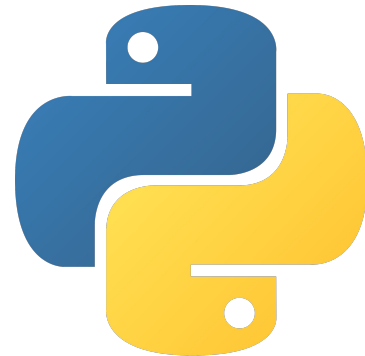
Apt-get: `sudo apt-get install python3.5 && apt-get install python-pip`

Yum: `sudo yum install python35 && yum -y install python-pip`

Windows

<https://python.org.br/instalacao-windows/>

1. **Ambiente:** Linux;
2. **Interpretação:** Via Terminal;
3. **Editor:** *Sublime Text 3* (pode ser qualquer um);
4. **Ambiente Integrado alternativo:** IDLE.



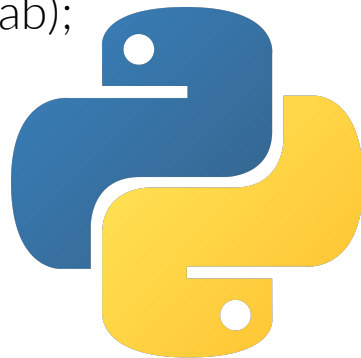
» SOBRE A LINGUAGEM

1. O Python não utiliza ponto e vírgula no fim das instruções, mas o ponto e vírgula pode ser usado para separar duas instruções em uma mesma linha;

Ex.:

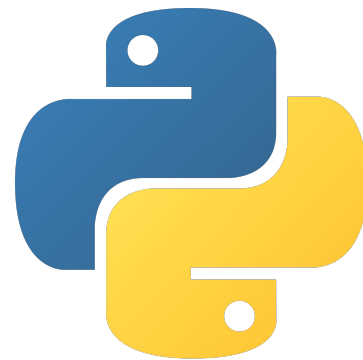
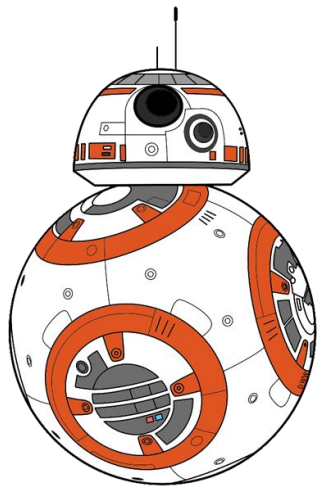
```
>>> print("Linha 1"); print("Linha2")
```

2. Linguagem baseada em indentação (indentação apenas com Tab);
3. TUDO em *python* é um objeto.



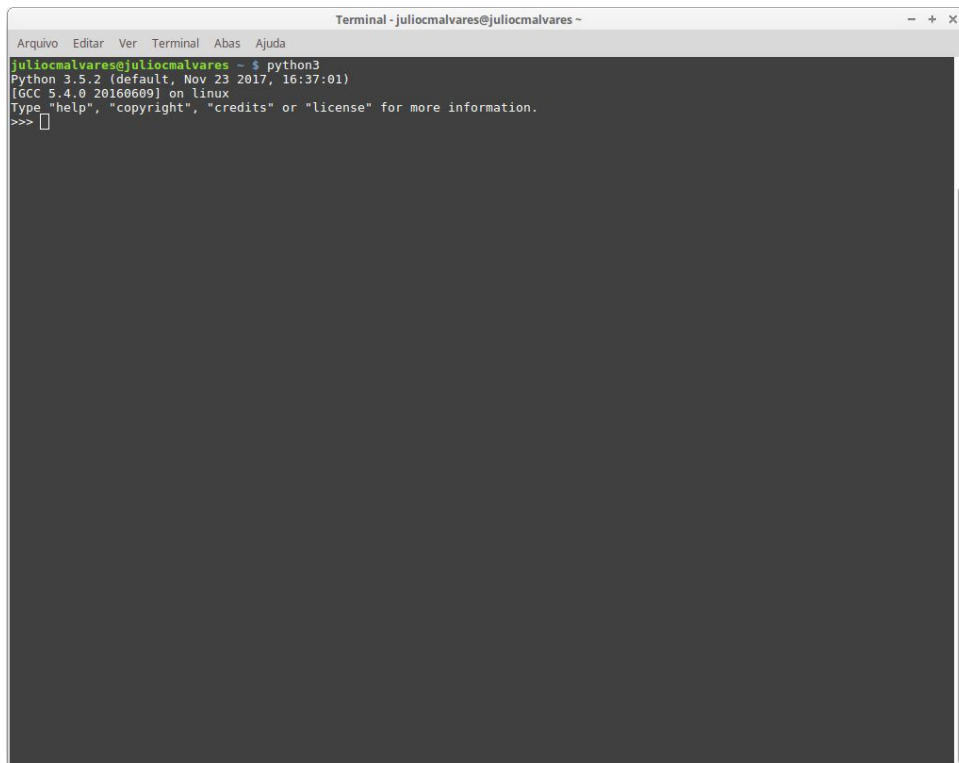
» SOBRE A LINGUAGEM

1. Extensão de arquivo “.py”;
2. Para executar um script, no terminal digite: “python3 script.py”.

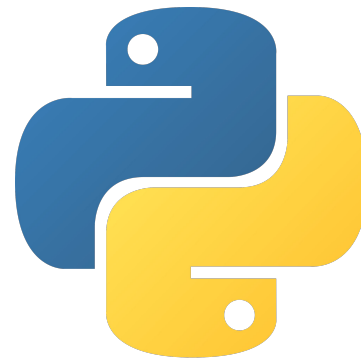


O AMBIENTE INTERATIVO NO LINUX

Basta abrir um terminal e digitar o comando “python3”

A screenshot of a Linux terminal window. The title bar reads "Terminal - juliocmalvares@juliocmalvares ~". The menu bar includes "Arquivo", "Editar", "Ver", "Terminal", "Abas", and "Ajuda". The terminal content shows the command "python3" being executed, followed by the Python 3.5.2 version information and the GCC compiler details. The prompt ">>>" is visible at the bottom of the terminal output.

```
Terminal - juliocmalvares@juliocmalvares ~
Arquivo  Editar  Ver   Terminal  Abas   Ajuda
juliocmalvares@juliocmalvares ~ $ python3
Python 3.5.2 (default, Nov 23 2017, 16:37:01)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```



» SOBRE A LINGUAGEM

- Comentários simples são feitos com a “#”;
- Comentários de várias linhas devem ser feitos com 3 aspas duplas (Também pode ser usado para atribuir um texto à uma variável).

```
1 #comentário simples
2
3 """
4 comentario de várias linhas
5 !!
6 !!
7 """
```



» VARIÁVEIS

- Nomeclatura de variáveis devem necessariamente começar por uma letra ou “_” (underscore);
- *Python* é case sensitive;
- Por convenção, o *python* utiliza nomenclatura de variáveis e funções em minúsculo, com separador underscore. Classes são nomeadas com a primeira letra de cada palavra maiúscula.

Ex.:

```
>>> nome_pessoa = “Julio”
```

```
>>> class Pessoa(object):
```



» VARIÁVEIS

- Têm tipo forte e dinâmico;
- Por a linguagem ser dinâmica, não quer dizer que sua tipagem é fraca. Não é possível somar *strings* com inteiros, por exemplo (Mas é possível converter facilmente);
- Mas você pode multiplicar *strings*. ヽ_(ツ)_/

Tente você mesmo:

```
>>> 'Curso de Python' * 10
```



» VARIÁVEIS

- Função `type()`;
- Conversão de tipos;

Ex.:

```
>>> a = 5
```

```
>>> b = str(a)
```

```
>>> print(type(b))
```

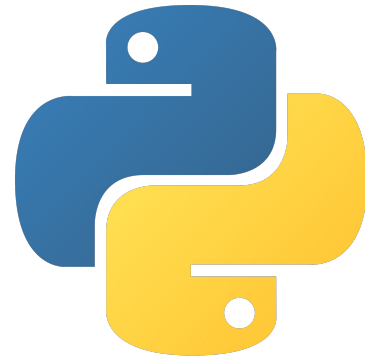
```
<class 'str'>
```



» ENTRADA DE DADOS

- Função `input()`;
- A função `input()` captura os dados do teclado como strings.
Convenção fazer *swap* para o tipo desejado;
- Sintaxe:

```
>>> a = int(input("Digite um número: "))
```



» OPERAÇÕES MATEMÁTICAS

Praticamente igual às outras linguagens!

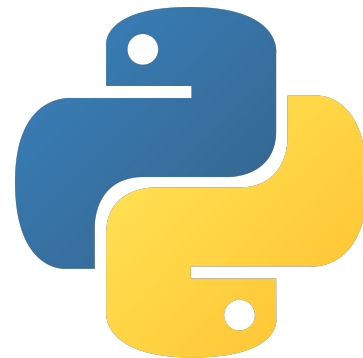
- Soma: +
- Subtração: -
- Multiplicação: *
- Divisão: /
- Potenciação e Radiciação: **
 - Para radiciação, podemos elevar um número ao inverso do segundo!
- Módulo da divisão: %



» OPERAÇÕES RELACIONAIS

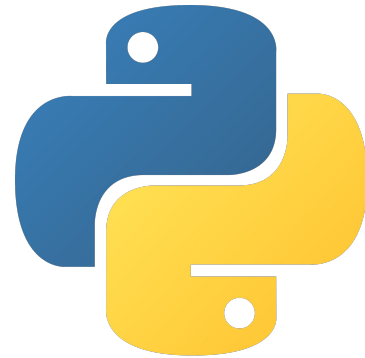
Praticamente igual às outras linguagens!

- Igual: ==
- Diferente: !=
- Menor que: <
- Maior que: >
- Menor igual/Maior igual: <= / >=



» OPERADORES LÓGICOS

- Operador e: *and*
- Operador ou: *or*



» LISTAS

Listas são conjuntos de objetos, como os *arrays* em C.

Ex.:

```
>>> a = [1,1]
```

```
>>> a[0] = 2 #Operação válida
```

```
>>> b = [1,1]
```

```
>>> c = a+b #c -> [2,1,1,1]
```



» LISTAS

Função *append(param)*: concatena o objeto passado como parâmetro no fim da lista.

Função *pop(param)*: Remove o item de índice passado por parâmetro (O parâmetro é o índice do item).

Ex.:

```
>>> a = [1,2,3]
```

```
>>> a.append(4) #Resultado: [1,2,3,4]
```

```
>>> a.pop(0) #Resultado: [2,3,4]
```



» TUPLAS

Tuplas são objetos com múltiplos valores (como uma lista), mas a principal característica de uma tupla é que seus elementos são imutáveis.

Ex.:

```
>>> a = (1,1)
```

```
>>> #a[0] = 2 Operação inválida
```

```
>>> b = (1,1)
```

```
>>> c = a+b #c -> (1,1,1,1)
```



» CONDIÇÕES

```
if condicao and condicao or condicao ... condicao:
```

```
    #código
```

```
elif condicao1 or condicao or condicao ... condicao:
```

```
    #codigo
```

```
elif condicao2:
```

```
    #codigo
```

```
else:
```

```
    #codigo
```



» LOOPS

O *python* tem 2 tipos de loops, o *for* e o *while*.

O While funciona exatamente como nas outras linguagens, mas podemos utilizar a cláusula *else* no fim.

```
while condicao:
```

```
    #codigo
```

```
else:
```

```
    #codigo
```

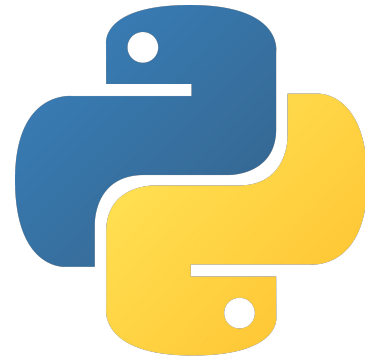


» LOOPS (FOR IN)

O *for* no *python* tem um funcionamento diferente de linguagens como o C.

No *python*, um *for* só pode ser feito sobre um objeto que seja iterável ou que se crie um iterável sobre ele.

Seu funcionamento consiste em criar um objeto local que irá assumir o valor de cada posição do objeto a ser iterado.



» LOOPS (FOR IN)

Ex.:

```
>>> lista = [1,2,3,4,5,6]
```

```
>>> for item in lista:
```

```
...     print(item)
```

O código acima funciona da seguinte forma: Um objeto iterável foi passado para o for, uma lista. O objeto “item” irá assumir o valor de cada posição da lista, iterando sobre todos.



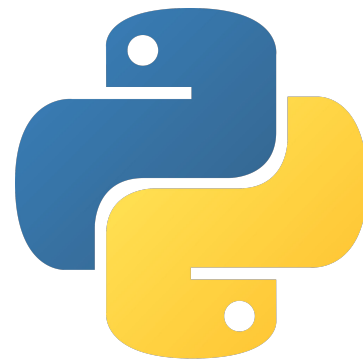
» LOOPS (FOR IN)

Função range()

Caso eu queira iterar uma determinada quantidade de vezes, é possível iterar sobre uma sequência criada pela função range().

Ex.:

```
>>> for i in range(20):  
...     print("Iteração: ", i)
```



» FUNÇÃO RANGE() COMPLETA

A função `range()` cria sequências. Em algum momento, será interessante criar sequências com início, fim e valor determinado, por exemplo, uma sequência de 0 a 20, de 2 em 2. (0,2,4 ... 20)

Para isso, podemos passar como parâmetro para a função.

Ex.: `lista = range(inicio, fim, valor)`

Caso passe apenas 1 parâmetro, será entendido que o mesmo é o fim da sequência e que o salto é de 1 em 1.



» *LIST COMPREHENSIONS*

Forma de criar uma lista em apenas uma linha, utilizando um for in “inline”.

Ex.:

```
>>> lista = [p for p in range(20)]
```



» *LIST COMPREHENSIONS*

Permite realizar condições dentro do for in. Por exemplo, desejo uma lista de 0 a 40 apenas com números múltiplos de 2.

Como fazer ?



» *LIST COMPREHENSIONS*

Resposta:

```
>>> lista = [p for p in range(40) if p % 2 == 0]
```

Qual a vantagem de usar *List Comprehensions*? Várias linhas em apenas 1.

Várias funções que manipulam listas geram *list comprehensions*, como a função *sorted()*, *map()* e outras.



VAMOS APLICAR UM POUCO

Question: Tenho uma lista “numbers” com os números {1,2,3,4,5} e desejo criar outra lista “square” com o quadrado de cada um deles.

```
>>> numbers = [1,2,3,4,5]
>>> square = []
>>> for item in numbers:
...     square.append(number*number)
>>> print(square)
```



VAMOS APLICAR UM POUCO

Agora vamos utilizar a função *map()*, que mapeia os elementos de uma lista.

```
>>> numbers = [1,2,3,4,5]
```

```
>>> square = list(map(lambda x: x*x, numbers))
```

```
>>> print(square)
```

Obs.: A função lambda foi usada para elevar os elementos ao quadrado. Não se preocupem, será explicado mais à frente.

Obs. 2: A função *map()* retorna um “mapa” da lista, do tipo *map*. Para retornar como lista, usamos a conversão de tipos!! Sempre recomendado fazer *swap*, o python tem tipo dinâmico, né?



VAMOS APLICAR UM POUCO

Agora vamos utilizar *list comprehensions*.

```
>>> numbers = [1,2,3,4,5]
```

```
>>> square = [p*p for p in numbers]
```

```
>>> print(square)
```

Bem mais legível e simples, não ?!



VAMOS APLICAR UM POUCO

Vamos complicar um pouco! Tenho a lista “numbers” com os elementos {1,2,3,4,5,6,7,8,9} e desejo o quadrado de todos os elementos menores que 5 na lista “square”.

```
>>> numbers = list(range(1,10))  
>>> square = list()  
>>> for i in numbers:  
...     if i < 5:  
...         square.append(i*i)  
>>> print(square)
```



VAMOS APLICAR UM POUCO

Agora vamos fazer o mesmo sem utilizar nenhum laço de repetição ou condição. Usaremos a função *filter()*, que filtra a lista de acordo com a condição passada por parâmetro.

```
>>> lista_aux = list(filter(lambda x:x<5,numbers))
```

A lista com os elementos menores que 5 estão em “lista_aux”, como os elevo ao quadrado sem utilizar *comprehensions* ou laço de repetição?



VAMOS APLICAR UM POUCO

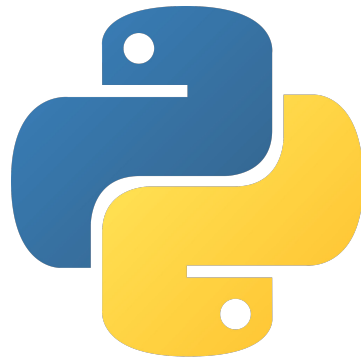
Podemos utilizar a função *map()* !!

```
>>> square = list(map(lambda x: x*x, lista_aux))
```

Fizemos muitos passos. Vamos “pythonizar” esse código !

```
>>> numbers = list(range(1,10))
```

```
>>> square = list(map(lambda x: x*x, list(filter(lambda x: x<5, numbers))))
```

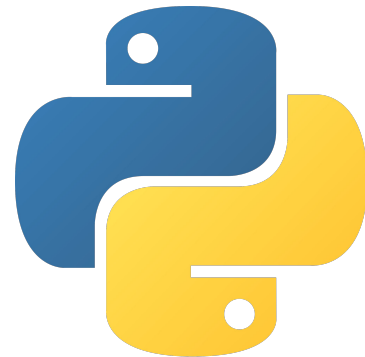


VAMOS APLICAR UM POUCO

Agora pythonizando com *list comprehensions*.

```
>>> numbers = list(range(1,10))
```

```
>>> square = [p*p for p in numbers if p < 5]
```



» STRINGS

- Strings são listas de caracteres concatenados imutáveis;
- Importante lembrar que são imutáveis.

Strings podem ser instanciadas com aspas simples ou aspas duplas.

Ex.:

```
>>> nome = 'Julio'
```

```
>>> nome2 = "Julio"
```



» STRINGS

- Já que são listas, funcionam como listas !

Ex.: Para acessar a segunda posição da string nome do slide anterior:

```
>>> print(nome[1])
```

u



» STRINGS

- E já que elas são listas, elas também podem ser iteradas como listas !

```
>>> for letra in nome:
```

```
...     print(letra)
```



» STRINGS

Funções importantes:

- `string.split()`
 - Quebra uma string em uma lista de strings de acordo com o parâmetro de quebra passado.

Parâmetros: `string.split(str_quebra)`

Ex.:

```
>>> text = "Senhor dos anéis é melhor que Harry Potter"
>>> lista_palavras = text.split(" ") #espaço vazio entre as aspas
>>> print(lista_palavras)

['Senhor', 'dos', 'aneis', 'é', 'melhor', 'que', 'Harry', 'Potter']
```



» STRINGS

Funções importantes:

- `string.replace()`
 - Strings são imutáveis para operações diretas, mas, as vezes precisamos trocar algum texto em uma string. Para isso usamos `replace`. Parâmetros: `string.replace(str_original, nova_str)`

Ex.:

```
>>> text = "Senhor dos anéis é melhor que Harry Potter"
>>> text = text.replace("melhor", "muito melhor")
>>> print(text)
```

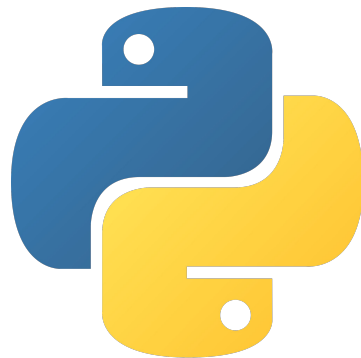


» STRINGS

- Funções importantes:
 - `string.format()`
 - Usada principalmente dentro da função `print()`
 - Serve para formatar a string de acordo com sua vontade;
 - A área a ser formatada na string deve conter chaves, ou seja, na string deve haver algo desse tipo “Julio {}”;
 - O parâmetro da função será inserido no lugar da chave.

Ex.:

```
>>>print("Python {} {} {}".format("é", "muito", "legal"))
```



» DICIONÁRIOS

- Estrutura de dados mutável com característica de disposição de elementos no formato *key:value*;
- Característica importante: A ordem de inserção dos elementos no dicionário não necessariamente é a ordem que será armazenada no dicionário;
- Listas são criadas com [], Tuplas com (), já os dicionários são criados com { }



» DICIONÁRIOS

- Pode ser usado para guardar dados correlacionados.
- Inserção no dicionário é feito explicitando a posição da *key* e atribuindo o *value*

Ex.:

```
>>> nome_idade = dict()
```

```
>>> nome['Julio'] = 21
```

O acesso a posição 'Julio' vai ser ineficaz, já que ela não existe, logo o python entende que deve ser criado uma nova posição e o *value* atribuído



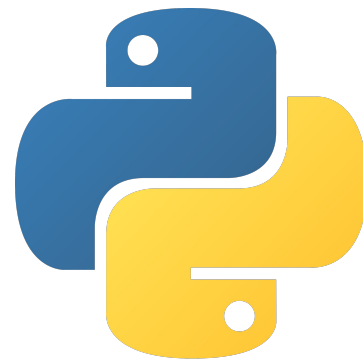
» DICIONÁRIOS

- Também pode ser criado de forma explícita.

Ex.:

```
>>> nome_idade = {'Julio':21, 'Fulano':12}
```

- Atento à sintaxe de *key:value*, usando sempre os dois pontos.



» DICIONÁRIOS

- Iterando sobre dicionários

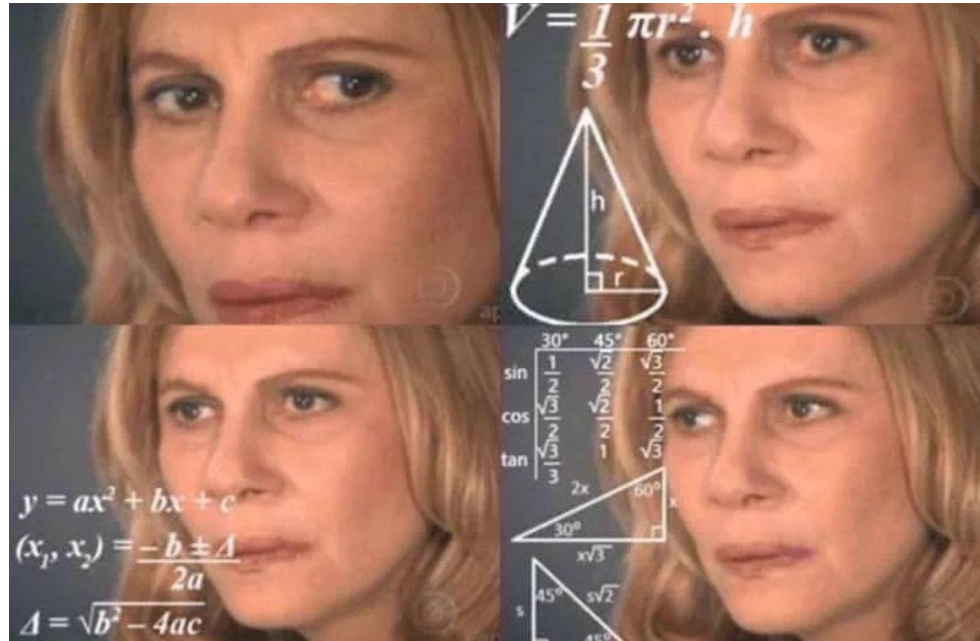
```
>>> for key in nome_idade:
```

```
...     print(nome_idade[key])
```



» FUNÇÕES, MÉTODOS E PROCEDIMENTOS

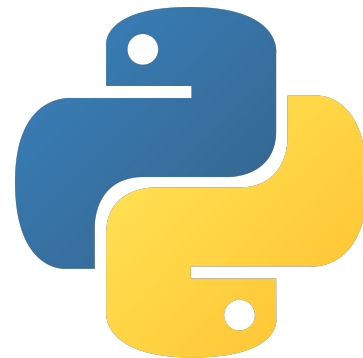
- São iguais, mas são diferentes! :D



» FUNÇÕES, MÉTODOS E PROCEDIMENTOS

- Uma função retorna um ou mais valores (Sim, o python retorna mais de um valor por função, se quiser. `<3 <3 <3 <3 <3`);
- Um procedimento não retorna valores, apenas faz processamentos;
- Um método é uma função ou procedimento de uma classe.

Fácil de lembrar, né?



» FUNÇÕES, MÉTODOS E PROCEDIMENTOS

- O funcionamento de funções em python são basicamente como nas outras linguagens. São blocos de códigos reservados que só são utilizados caso sejam invocados.
- A invocação de uma função faz com que o *program counter* seja deslocado para a função, seu processamento é feito e logo após o *program counter* retorna para a linha abaixo da função (caso a mesma não invoque outras funções, aí funciona da mesma forma).

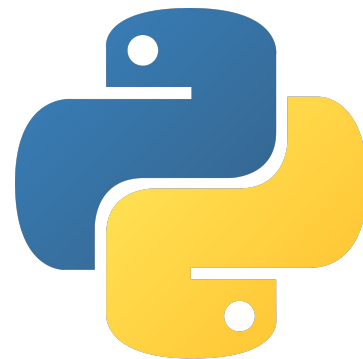


» FUNÇÕES, MÉTODOS E PROCEDIMENTOS

- Para definirmos uma função/procedimento, utilizamos a palavra reservada “def” seguida do nome da função, abrimos e fechamos parênteses e dois pontos. Os parâmetros ficam dentro dos parênteses. (Métodos veremos na próxima aula)

Ex.:

```
>>>def soma(x1,x2):  
...     return x1+x2  
>>> soma(1,2)
```



» FUNÇÕES, MÉTODOS E PROCEDIMENTOS

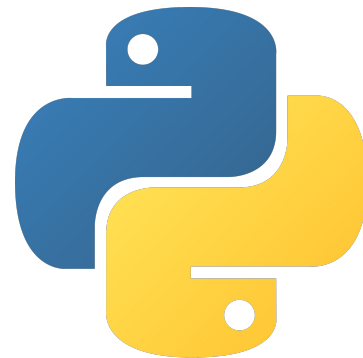
Exemplo de funções/procedimentos.

```
>>> def print_msg(msg):  
...     print("A msg é: {}".format(msg))  
>>> print_msg("Nunca mais programo em C")
```



» FUNÇÕES, MÉTODOS E PROCEDIMENTOS

```
>>> def maior(lista):  
...     maior = -1  
...     for elem in lista:  
...         if elem > maior:  
...             maior = elem  
...     return maior  
>>> lista = list(range(1, 100))  
>>> print(maior(lista))
```



» FUNÇÕES, MÉTODOS E PROCEDIMENTOS

» PARÂMETROS DEFAULT

- São parâmetros pré-definidos na codificação da função;
- Se você não explicitar o valor do parâmetro, o default é invocado.

Ex.:

```
>>> def dados(nome = 'Julio', idade = 21):  
...     print("Nome: {}\nIdade: {}".format(nome, idade))  
  
>>> dados("João", 23)  
  
>>> dados()  
  
>>> dados("Maria", 50)
```



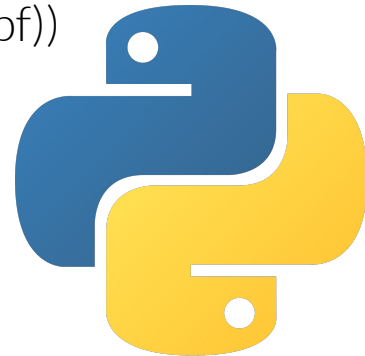
» FUNÇÕES, MÉTODOS E PROCEDIMENTOS

» PARÂMETROS NOMEADOS

- Serve para explicitar qual parâmetro está passando na invocação da função. (Bom para funções com muitos parâmetros)
- Podem ser chamados fora de ordem.

Ex.:

```
>>> def dados(nome='Julio', idade=21, cpf='aaa.bbb.ccc-xx'):  
...     print("Nome: {}\nIdade: {}\nCpf: {}".format(nome, idade, cpf))  
>>> dados(cpf='111.222.333-44', idade = 32, nome = 'Joao')  
>>> dados(cpf = '123.321.456-20')
```



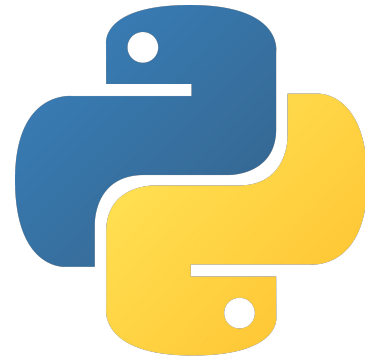
» FUNÇÕES, MÉTODOS E PROCEDIMENTOS

» RETORNO DE VALORES.

- Uma função pode retornar um resultado após o processamento.
 - Palavra reservada *return*

```
>>> def soma(x1, x2):
```

```
...     return x1+x2
```



» FUNÇÕES, MÉTODOS E PROCEDIMENTOS

» RETORNO DE VALORES MÚLTIPLOS.

- Uma função pode retornar vários resultados após o processamento.
 - Palavra reservada *return* juntamente com todas as variáveis.

```
>>> def soma(x1, x2):
```

```
...     return x1+x2, x1, x2 #Retorno a soma, o primeiro valor e o  
segundo
```



» FUNÇÕES, MÉTODOS E PROCEDIMENTOS

» RETORNO DE VALORES MÚLTIPLOS.

- Quando invocamos uma função de múltiplo retorno, temos duas situações:
 - atribuição em variáveis múltiplas, atribuição em variável única.

```
>>> soma, x1, x2 = soma(x1,x2)
```

```
>>> dados = soma(x1,x2)
```

- No primeiro caso, cada variável vai receber um retorno;
- No segundo caso, a variável “dados” vai receber uma tupla com o conjunto de retornos.



» FUNÇÕES, MÉTODOS E PROCEDIMENTOS

» FUNÇÕES VARIÁDICAS

- Funções que podem receber 0 ou vários parâmetros.

```
>>> def func(*args, **kwargs):  
...     pass
```

- **args* é a definição de uma lista de parâmetros a passar para a função;
- ***kwargs* é a definição de uma lista de parâmetros nomeados, dados em formato de dicionário.

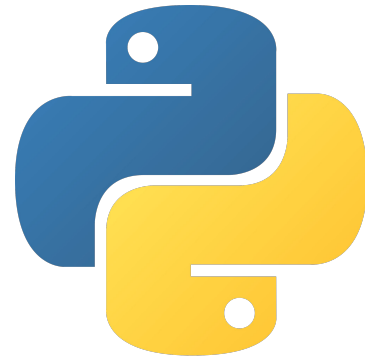


» FUNÇÕES, MÉTODOS E PROCEDIMENTOS

» FUNÇÕES VARIÁDICAS

- A função recebe uma lista de argumentos e a transforma em uma tupla na variável.

```
>>> def lista_args(*lista):  
...     print(lista)  
  
>>> lista_args(1,2,3,4,5,6,7,8,9)  
  
>>> lista_args()  
  
>>> lista_args("um", "dois", "três")
```



» FUNÇÕES, MÉTODOS E PROCEDIMENTOS

» FUNÇÕES VARIÁDICAS

- Parâmetro em forma de lista associativa. (Dicionário)

```
>>> def lista_param(**dicionario):
```

```
...     print(dicionario)
```

```
>>> lista_param(a=1,b=2,c=3,d=4)
```



» LEITURA E ESCRITA DE ARQUIVOS.

- Criar um objeto e invocar a função `open(nome_arq, tipo);`

Ex.:

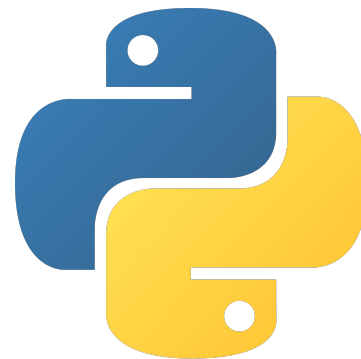
```
>>> f = open("entrada.txt", "r") #tipo r é para apenas leitura.
```

- Funções de leitura:
 - `f.read()` lê todo o conteúdo do arquivo e joga em um objeto;
 - `f.readlines()` lê uma linha por vêz.



» LEITURA E ESCRITA DE ARQUIVOS.

- Funções de escrita:
 - `f.write(string)` escreve a string no arquivo.
- Função de manipulação de ponteiro de escrita:
 - `f.seek(position)` posição sendo para onde quer apontar.
- Funções de abertura e fechamento de arquivo:
 - `f.open(nome_arq, tipo)`
 - `f.close()`



» LEITURA E ESCRITA DE ARQUIVOS.

- Convenção do Python quanto a abertura de arquivos:
 - Utilizar o método *with*

```
>>> with open("entrada.txt", "r") as f:
```

```
...     texto = f.read()
```

```
...     #todas as operações de leitura do arquivo serem realizadas aqui  
dentro.
```

```
>>> f.close() #sempre fechar o arquivo no final. MUITO IMPORTANTE
```



» LEITURA E ESCRITA DE ARQUIVOS.

Modos de abertura de arquivos da função *open()*

- “r” : Apenas leitura, ponteiro no início do arquivo;
- “r+” : Leitura e escrita, ponteiro no início do arquivo;
- “w” : Apenas escrita, apaga todo conteúdo, ponteiro no início;
- “w+” : Leitura e escrita, apaga conteúdo, ponteiro no início;
- “a” : Apenas escrita, não apaga, ponteiro no fim;
- “a+” : Leitura e escrita, não apaga, ponteiro no fim.



» MÓDULOS

- De uma forma mais rústica, um módulo é um arquivo .py qualquer. O mesmo pode ser invocado em outros módulos para fazer algum processamento.
- Assim que um módulo é invocado, todas as funções e variáveis globais do módulo invocado serão inseridos no módulo que o invocou.



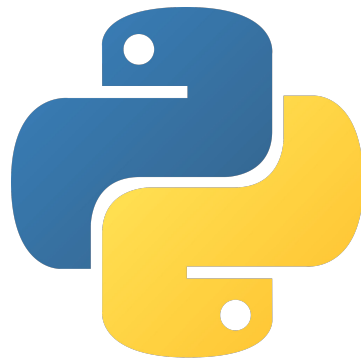
» MÓDULOS

- Importação de módulos:

>>> import modulo #importa o modulo completo

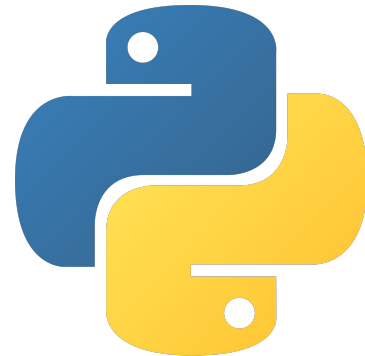
*>>> from modulo import * #importa o módulo completo*

>>> from modulo import soma #importa apenas a função soma



» REFERÊNCIAS E CÓPIAS EM PYTHON

- Sabe quando a Tia Doralice falou pra você passar o parâmetro por referência para uma função, ou pediu pra você copiar os valores de um vetor ? Como que faz em python ??
- Listas sempre são manipuladas por referências. Qdo quiser manipular um valor por referência, utilize um operador global ou passe o parâmetro como uma lista.



» REFERÊNCIAS E CÓPIAS EM PYTHON

- Operador global:

```
>>> msg = "hello world"
```

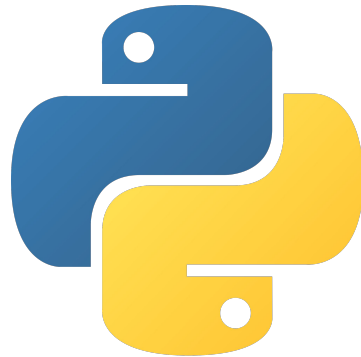
```
>>> def func():
```

```
...     global msg
```

```
...     msg = "hello world global"
```

```
...     print(msg)
```

```
>>> func()
```



» REFERÊNCIAS E CÓPIAS EM PYTHON

- Como as listas sempre são manipuladas por referência, copiar uma lista para outra variável só vai mudar o nome da mesma.

```
>>> a = [1,2,3]
```

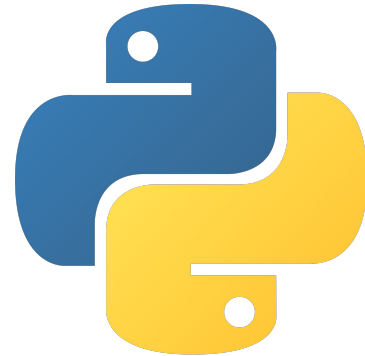
```
>>> b = a
```

```
>>> b[0] = 5
```

```
>>> print(a)
```

- Slicing de listas:

```
>>> b = a[:] # sintaxe: a[posição inicial: posição final - 1]
```



» REFERÊNCIAS E CÓPIAS EM PYTHON

- Também podemos usar a função biblioteca *copy*

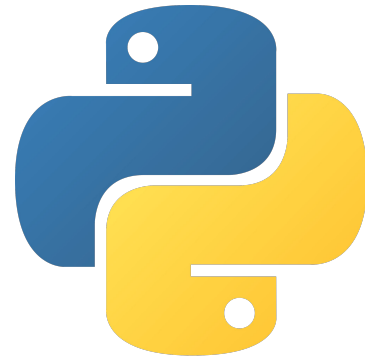
```
>>> from copy import *
```

```
>>> a = [1,2,3]
```

```
>>> b = copy(a)
```

```
>>> b[0] = 5
```

```
>>> print(a, b)
```



» FUNÇÕES INTERESSANTES.

- Listas.

- `lista.insert(i, x)` sendo `i` o índice e `x` o elemento a ser inserido;
- `lista.remove(x)` remove o primeiro item `x` encontrado na lista;
- `lista.index(x)` retorna o primeiro índice do valor `x` encontrado;
- `lista.sort()` aplica um algoritmo de ordenação na lista;
- `lista.reverse()` aplica um algoritmo de ordenação inversa na lista.
- `set()` cria um conjunto sem duplicatas de uma lista.
- Função `sorted()` gera um *list comprehension* com a lista passada como parâmetro ordenada.



ORIENTAÇÃO A OBJETOS COM PYTHON

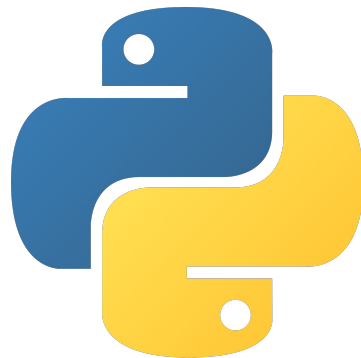
JAVA NUNCA MAAAIS !!!



» PROGRAMAÇÃO ORIENTADA A OBJETOS

- Introduzida por Alan Kay com a linguagem SmallTalk, tendo como objetivo se aproximar ao máximo do mundo real;
- Rendeu-lhe o prêmio Turing em 2003 e o prêmio Kyoto de 2004;
- Matemático, Biólogo e Cientista da Computação;
- Criou o Laptop (notebook);
- Criou as interfaces gráficas modernas (GUI).

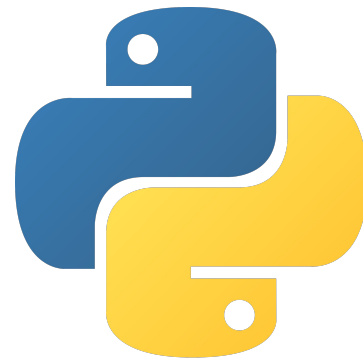
(O cara é brabo)



» PROGRAMAÇÃO ORIENTADA A OBJETOS

“O problema de linguagens orientadas a objetos é que você tem todo esse ambiente implícito que elas levam consigo. Você queria uma banana, mas o que você conseguiu foi um gorila segurando a banana e uma selva completa” - Joe Armstrong

- O seu programa deve ser extensível.

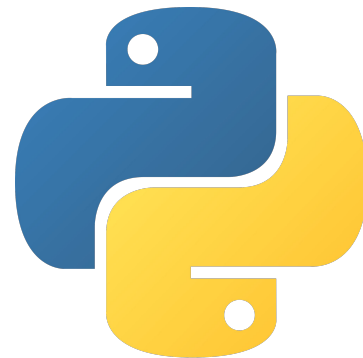


» PROGRAMAÇÃO ORIENTADA A OBJETOS

```
>>> a = int(2) #Atribuição de variáveis também pode ser feita assim.
```

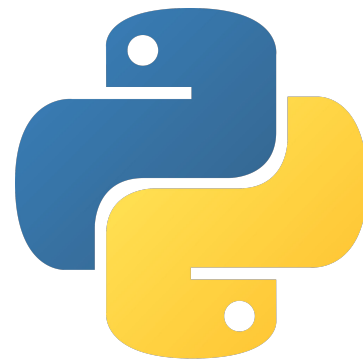
```
>>>a.__sizeof__() #comando sizeof() do C. Obs.: tem 2 underscore
```

- O resultado da operação deve ser 28 (bits);
- Um número inteiro tem apenas 16 bits, logo, os outros 12 bits são métodos e outros atributos da classe int.
- Conclusão: “Grandes poderes vem com grandes responsabilidades” - Tio Ben (2002).



» PROGRAMAÇÃO ORIENTADA A OBJETOS

- Classes
- Objetos
- Atributos → Dados
→ Métodos
- Herança
- Polimorfismo
- Sobrecarga de Operadores
- Encapsulamento
- MetaClasses
- Exceções



» CLASSES

- São os “moldes” dos objetos.
- Definidos com a palavra reservada *class* (como em outras linguagens);

```
>>> class Pessoa(object): #como vimos, classes primeira letra maiúscula  
...     pass
```

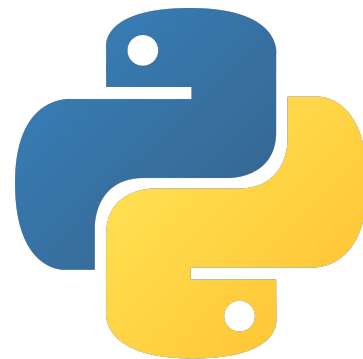
- Entre parênteses, são as classes na qual a minha classe herda, toda classe em python herda da classe *object*;
- Python suporta Herança múltipla. (veremos)



» DADOS

- Atributos de dados não precisam ser explicitados na classe;
- Normalmente atribuídos no método *init*;

```
>>> class Pessoa(object):  
...     nome = "  
...     idade = None  
...     def __init__(self, nome, idade):  
...         self.nome = nome  
...         self.idade = idade
```



» DADOS

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



» MÉTODOS

- Funções ou procedimentos dentro de classes;
- Definidos da mesma forma como vimos na última aula;
- Para referenciar ao próprio objeto, o python utiliza a palavra reservada *self*. O mesmo deve ser passado como parâmetro e deve ser obrigatoriamente o primeiro. (Na chamada da função não é necessário passá-lo como parâmetro.



» MÉTODOS

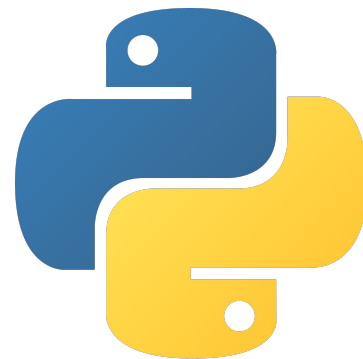
```
>>> class Pessoa(object):  
...     def get_nome(self):  
...         return self.nome  
...     def set_nome(self, nome):  
...         self.nome = nome  
  
>>> pessoa = Pessoa()  
  
>>> pessoa.set_nome('Julio')  
  
>>> print(pessoa.get_nome())
```



» DUNDER METHODS

- Métodos “dunder” são métodos especiais que definem interfaces sobre as classes;
- Alguns métodos dunder: Construtores, somadores, multiplicadores, destrutores, comparadores, string;
- Podem ser sobrecarregados e/ou criados em qualquer classe
- São definidos com 2 underscore antes e depois do seu nome

```
>>> class Pessoa(object):  
...     def __init__(self, nome, idade): #construtor  
...         self.nome = nome  
...         self.idade = idade
```



» DUNDER METHODS

- Para saber quais os atributos temos em uma classe, podemos utilizar o comando `help(class)` no ambiente interativo;
- O resultado do `help` é um *doc* sobre a classe;
- O comando `dir(class)` retorna uma lista com os atributos de uma classe;

```
>>>help(list)
```

```
>>>dir(list)
```



» DUNDER METHODS

Binary Operators

Operator

+

-

*

//

/

%

**

<<

>>

&

^

|

Method

object.__add__(self, other)

object.__sub__(self, other)

object.__mul__(self, other)

object.__floordiv__(self, other)

object.__truediv__(self, other)

object.__mod__(self, other)

object.__pow__(self, other[, modulo])

object.__lshift__(self, other)

object.__rshift__(self, other)

object.__and__(self, other)

object.__xor__(self, other)

object.__or__(self, other)



» DUNDER METHODS

Extended Assignments

Operator

+=

-=

*=

/=

//=

%=

**=

<<=

>>=

&=

^=

|=

Method

object.__iadd__(self, other)

object.__isub__(self, other)

object.__imul__(self, other)

object.__idiv__(self, other)

object.__ifloordiv__(self, other)

object.__imod__(self, other)

object.__ipow__(self, other[, modulo])

object.__ilshift__(self, other)

object.__irshift__(self, other)

object.__iand__(self, other)

object.__ixor__(self, other)

object.__ior__(self, other)



» DUNDER METHODS

Unary Operators

Operator

-
+
abs()
~
complex()
int()
long()
float()
oct()
hex()

Method

object.__neg__(self)
object.__pos__(self)
object.__abs__(self)
object.__invert__(self)
object.__complex__(self)
object.__int__(self)
object.__long__(self)
object.__float__(self)
object.__oct__(self)
object.__hex__(self)



» DUNDER METHODS

Comparison Operators

Operator	Method
<	object.__lt__(self, other)
<=	object.__le__(self, other)
==	object.__eq__(self, other)
!=	object.__ne__(self, other)
>=	object.__ge__(self, other)
>	object.__gt__(self, other)



» CHAMANDO MÉTODOS

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

- Como instanciar e chamar os métodos da classe acima ?



» CHAMANDO MÉTODOS

```
>>> p = Pessoa('Julio')
```

```
>>> print(p.nome)
```

```
>>> p.resetar() #observem que o parâmetro self não é passado
```

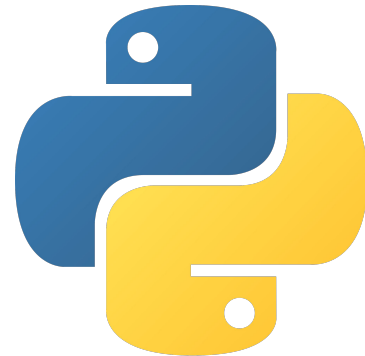
```
>>> print(p.nome)
```

```
>>> p = Pessoa('Julio')
```

```
>>> print(p.nome)
```

```
>>> Pessoa.resetar(p) #aqui o parâmetro é passado
```

```
>>> print(p.nome)
```



» CHAMANDO MÉTODOS

- Provando !

```
>>> int.__add__(3,2) #proprio objeto é o 3, seria o self
```

```
>>> 3.__add__(2) #dar um espaço dps do 3. (sem espaço o python  
entende que é um float)
```



» HERANÇA

- Toda classe do python utiliza herança;
- Toda classe do python herda naturalmente da classe *object*;

```
>>> class MinhaClasse(object):
```

```
...     pass
```

↑
Super Classes



» HERANÇA

- Sobrecarregando o construtor
- Comando `super()`

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



» HERANÇA

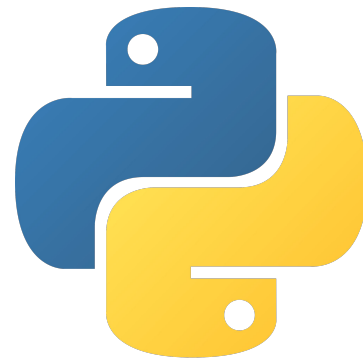
```
>>> class PessoaFisica(Pessoa):  
...     def __init__(self, nome, idade, cpf):  
...         super(PessoaFisica, self).__init__(nome, idade)  
...         self.cpf = cpf
```

chamando a classe pai



» HERANÇA MÚLTIPLA

- Uma classe herdar de 2 ou mais classes;
- Gera alto acoplamento entre as classes pai/filho, logo, qualquer mudança nas classes pai gera problemas nas classes filho;
- Pense em heranças como um corpo. As classes pai são os ossos e as filhas os músculos. Qualquer problema nos ossos, acarretará problema nos músculos.



» HERANÇA MÚLTIPLA

- Method Resolution Order:
 - Algoritmo do python que decide qual método de qual classe será executado primeiro (em ordem)

```
>>> int.mro()
```

```
[<class 'int'>, <class 'object'>]
```

- Os métodos da classe *int* tem prioridade sobre os da classe *object*.



» HERANÇA MÚLTIPLA

Funcionamento:

- Cria um grafo com a hierarquia de classes;
- Busca em profundidade começando pela classe do invocou o método e em largura da esquerda pra direita.



» HERANÇA MÚLTIPLA

- Vamos utilizar o arquivo “relogio.py” para explicar herança múltipla;
- Sintaxe do *super()* para múltiplos pais:

```
>>> super(ClasseAtual, self).method_do_pai(args)
```



» HERANÇA MÚLTIPLA

- Opte sempre por não utilizar o *super()* para diversas classes. A complexidade irá aumentar muito !
- Utilize sempre chamada da função + method

```
>>> Relogio.__str__() + " " + Calendario.__str__()
```

é melhor que

```
>>> super(CalendarioRelogio,self).__str__()
```

Why ?



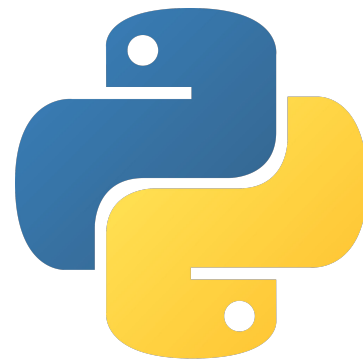
» HERANÇA MÚLTIPLA

- Uma boa utilização de herança múltipla é na chamada classe Mixin;
- São classes que manipulam atributos de OUTRAS classes;
- São definidas utilizando herança múltipla;
- Regra 1 da classe Mixin: Não se pode falar da classe Mixin;
- Regra 2 da classe Mixin: Não se pode falar da classe Mixin;
- Regra 3 da classe Mixin: Não devem ser instanciadas;
- Regra 4 da classe Mixin: Não devem ser estendidas;
- Se beneficia do alto acoplamento das classes para modularizar.



» HERANÇA

- Considerações sobre herança:
 - Na dúvida, não utilize;
 - Se você souber muito bem o que está fazendo, é um ótimo recurso, em caso de dúvida, a herança vai piorar seu problema;
 - Opte por composição de classes.



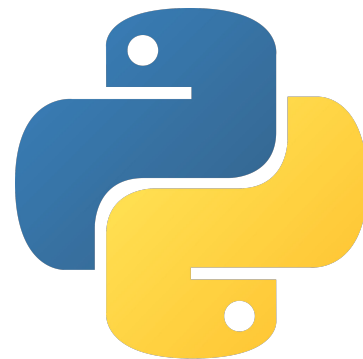
» POLIMORFISMO

- Famoso poliformismo!
- É uma característica que define que métodos podem ter comportamentos diferentes, ou seja, métodos podem se comportar polimorficamente.
- Quem decide a forma é o objeto que invocou o método.



» POLIMORFISMO

- Imaginemos que temos a classe “Cachorro” e as classes “PitBull” e “Pinscher” e que as duas últimas herdem da primeira;
- Cachorros latem, mas de formas e intensidades diferentes;
- Digamos que o pitbull lata da forma “AAU”;
- Digamos que o pinscher lata da forma “Tremedeira tremedeira auau tremedeira”;
- Teremos então:



» POLIMORFISMO

```
>>> class Cachorro(object):  
...     def __init__(self, nome):  
...         self.nome = nome  
...     def latir(self):  
...         return "AU"  
  
>>> class PitBull(Cachorro):  
...     def latir(self):  
...         return "AAU"
```



» POLIMORFISMO

```
>>> class Pinscher(Cachorro):  
...     def latir(self):  
...         return 'Tremedeira tremedeira auau tremedeira'  
  
>>> dog1 = PitBull('Bob')  
>>> dog2 = Pinscher('Demonio')  
>>> print(dog1.latir())  
>>> print(dog2.latir())
```



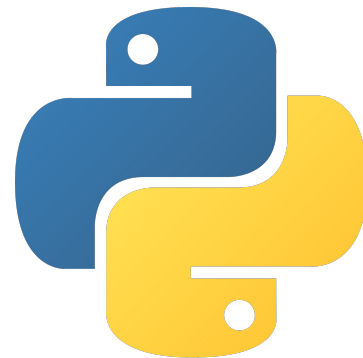
» POLIMORFISMO

- Então, o polimorfismo em python condiz com sobrescrever de métodos;
- Lembrando, são definidos pelo OBJETO que invoca o método;
- Python é inteiramente polimórfico.



» SOBRECARGA DE OPERADORES

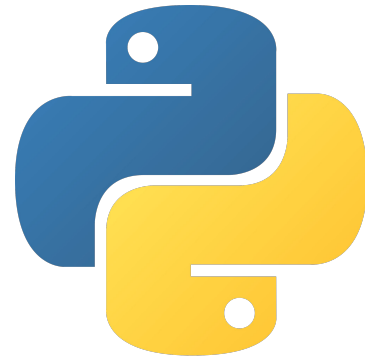
- A sobrecarga de operadores serve para mudar o comportamento de operadores como o “+”, “-”, “*”, etc;
- É feita sobrecarregando os *Dunder methods* do tipo do operador; (Olhar tabelas nos slides anteriores);
- O arquivo “operadores.py” tem exemplos de sobrecargas !
- Não abuse da sobrecarga dos operadores.



» ENCAPSULAMENTO EM DADOS

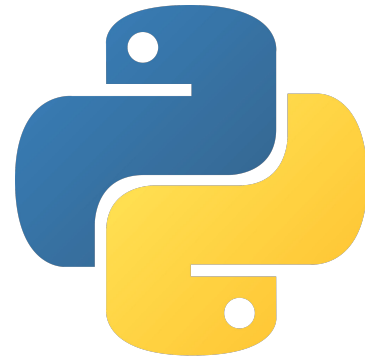
- Em python existem apenas dois tipos de atributos de dados: Públicos e privados;
- Para definir atributos públicos, basta nomeá-los normalmente;
- Para definir atributos privados, seu nome deve conter dois underscore como prefixo;

```
>>> class Pessoa():  
...     def __init__(self, nome, x):  
...         self.nome = nome #público  
...         self.__x = x #privado
```



» ENCAPSULAMENTO “REAL”

- O encapsulamento em python condiz com as regras que serão submetidas aos seus atributos de dados;
- comando `@property` define getter;
- comando `@atr.setter` define setter;
- Ambos os métodos implementados devem ter o nome do atributo;
- No arquivo “encapsulamento.py” há um exemplo.



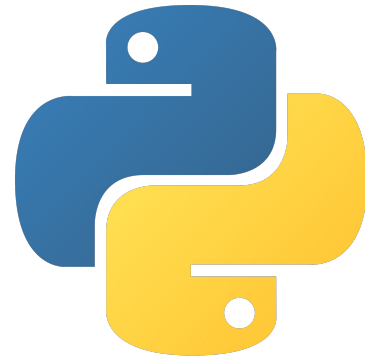
» ENCAPSULAMENTO “REAL”

- Vantagem:
 - Se você tem uma classe principal e diversas classes filhas que herdam da principal, se as suas classes filhas fazem acesso direto aos atributos da classe pai e você quiser encapsular os atributos da classe pai, não há necessidade de manutenção de código das classes filhas.



» CLASSES ABSTRATAS

- Abstract class são classes para definir classes;
- Seus métodos , por convenção, não implementam nada e são chamados de abstract methods;
- Utilizamos o módulo abc chamando ABCMeta e abstractmethod;
- O arquivo abstract.py será usado para explicação.



» CLASSMETHOD E STATICMETHOD

- Class methods são métodos que referenciam a uma classe;
- Static methods são funções dentro de uma classe, ou seja, eles não referenciam nem ao objeto nem à classe, apenas fazem algum processamento dentro da classe;
- São definidos com os decorators `@classmethod` e `@staticmethod`;
- O arquivo `static.py` exemplifica.



» EXCEÇÕES

- Exceções são erros de execução no código, como uma divisão por 0, por exemplo;
- Códigos são muito frágeis;
- Exceções também são objetos;
- Todas as exceções em python herdam da classe *BaseException*;
- Exceptions famosas: *SyntaxError*, *ZeroDivisionError*, *IndexError*, *TypeError*, etc.



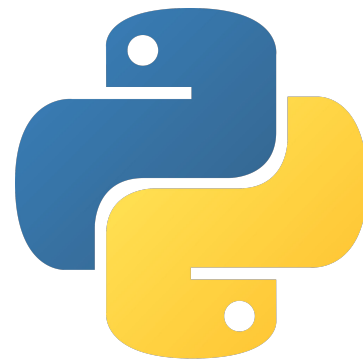
» EXCEÇÕES

- Para lançar uma exception, utilizamos a palavra reservada *raise* e o nome da exception;

```
>>> def f():
```

```
...     raise Exception('Lançando uma exception')
```

- O *raise* vai cortar a execução do seu código.



» MANIPULANDO EXCEÇÕES

- Para tratar uma exception, utilizamos a cláusula *try...except*;
- O *Try except* captura a exceção e podemos tratá-la, logo a execução do código não é interrompida;
- Sintaxe:

```
>>> try:
```

```
...     #código que pode lançar uma exception
```

```
... except ExceptionType:
```

```
...     #tratamento
```

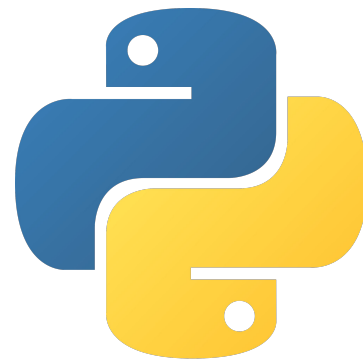
```
... except ExceptionType:
```

```
...     #tratamento
```



» CRIANDO NOSSAS EXCEÇÕES

- Para criar nossas exceções, basta criar uma classe que herde de *Exception* e fazer os tratamentos dentro dela;
- O arquivo `exceptions.py` contém exemplos de exceções.



BIBLIOTECAS CIENTÍFICAS

Manipulando dados de forma correta e fácil !!



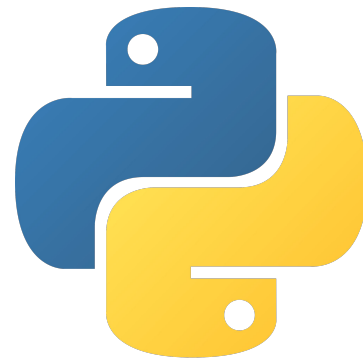
» SUMÁRIO

- Numpy
- Matplotlib
- Pandas
- NLTK (Natural Language ToolKit)
- SKLearn (Science Kit)



» NUMPY

- Parte do pacote *Scipy*;
- Biblioteca científica para manipulação de dados numéricos;
- Suporte poderoso para *N-dimensional arrays*;
- Funções de manipulação matemática sofisticadas e completas;
- Integração com C/C++, Fortran e MATLAB;
- Poderosos módulos de álgebra linear, transformada de *fourier* e números randômicos.



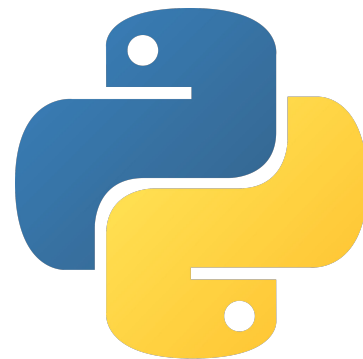
» NUMPY

- Projeto *source code* e de desenvolvimento comunitário pela *Python Software Foundation*;
- Amplamente utilizada para software científico;
- Altamente integrada com outras bibliotecas científicas.
- Documentação completa: <https://www.numpy.org>



» NUMPY

- Instalação:
 - pip: *sudo pip3 install numpy*
 - apt: *sudo apt-get install python-numpy*



» NUMPY

- Importar o módulo *numpy* e renomeá-lo para *np*; (Apenas para facilitar mesmo)

```
>>> import numpy as np
```



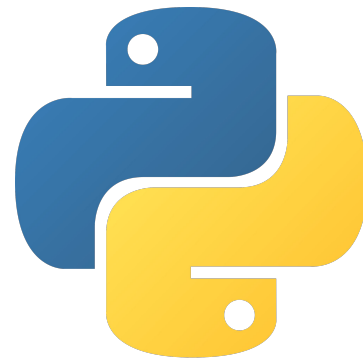
» NUMPY - ARRAYS

- Um *np.array* é basicamente uma lista, podendo ela ter N dimensões;
- Vamos considerar que o módulo está carregado e renomeado em todos os exemplos de código.

```
>>> a = [p for p in range(10)]
```

```
>>> array = np.array(a)
```

```
>>> print(array)
```



» NUMPY - ARRAYS

- Função *reshape()*:
 - faz a divisão entre as dimensões da matriz;
 - recebe uma lista de argumentos, sendo cada um uma dimensão da matriz;
 - *np.array().reshape(linhas, colunas)*;
 - *np.array().reshape(profundidade, linhas, colunas)*

```
>>> a = np.array([p for p in range(10)]).reshape(2,5)
```

```
>>> print(a)
```

```
>>> print(a.shape)
```

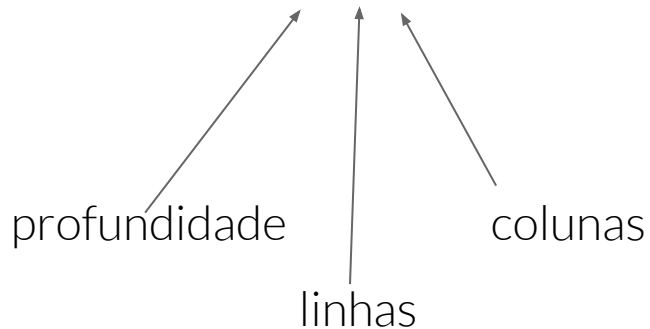


» NUMPY - ARRAYS

- Multidimensional:

```
>>> a = np.array([p for p in range(30)]).reshape(3,2,5)
```

```
>>> print(a)
```



» NUMPY - ARRAYS

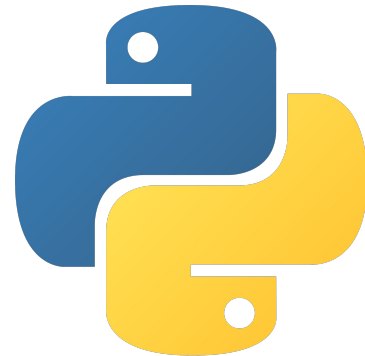
- Outra forma de se fazer:

```
>>> a = np.arange(0,20) #Crio um array de 20 elementos
```

```
>>> mat = np.reshape(a, (5,4))
```

- Para acessar um elemento:

```
>>> mat[2,2]
```



» NUMPY - ARRAYS

- Copiando *numpy arrays*
 - As operações normais de listas funcionam com *numpy arrays*, menos a cópia, ou seja, se fizer cópia de um *array* mesmo utilizando *slicing* as mudanças refletirão em todos;
 - Como fazer? método *array.copy()*;

```
>>> a = np.array([1,2,3,4])
```

```
>>> b = a.copy()
```



» MATPLOTLIB

- Biblioteca de visualização de dados;
- Gera gráficos com qualidade extremamente alta;
- Extensão da biblioteca NumPy;
- Vamos utilizar o módulo *pyplot*;
- Instalação: *sudo pip3 install matplotlib*



» MATPLOTLIB

- Exemplo simples:

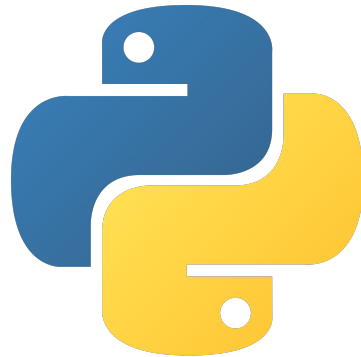
```
>>> import matplotlib.pyplot as plt
```

```
>>> import numpy as np
```

```
>>> salario = np.array([100,200,300,400,500,150])
```

```
>>> plt.plot(salario)
```

```
>>> plt.show()
```



» MATPLOTLIB

- Customizando gráficos
 - vamos assumir o exemplo anterior ainda carregado.

```
>>> plt.plot(salario, c='red') #parametro c de color da linha
```

```
>>> plt.show()
```

```
>>> plt.plot(salario, c='black', ls='--') #ls muda o tipo da linha
```

```
>>> plt.show()
```

```
>>> plt.plot(salario, ls='--', marker='o') #marcador de pontos
```

```
>>> plt.show()
```



» MATPLOTLIB

- Customizando gráficos

```
>>> salario2 = np.array([50,100,150,200,600])
```

```
>>> plt.plot(salario, c='black', marker='o', ls='--', label='Salário 1')
```

```
>>> plt.plot(salario2, c='red', marker='o', ls='--', label='Salário 2')
```

```
>>> plt.legend()
```

```
>>> plt.show()
```



» MATPLOTLIB

- Customizando gráficos

```
>>> plt.plot(salario, c='black', marker='o', ls='--', label='Salário 1')
```

```
>>> plt.plot(salario2, c='red', marker='o', ls='--', label='Salário 2')
```

```
>>> plt.legend()
```

```
>>> plt.ylabel('Salário (R$)')
```

```
>>> plt.xlabel('Tempo (meses)')
```

```
>>> plt.show()
```



» MATPLOTLIB

- Customizando gráficos

```
>>> plt.plot(salario, c='black', marker='o', ls='--', label='João')
```

```
>>> plt.plot(salario2, c='red', marker='o', ls='--', label='José')
```

```
>>> plt.legend()
```

```
>>> plt.ylabel('Salário (R$)')
```

```
>>> plt.xlabel('Tempo (meses)')
```

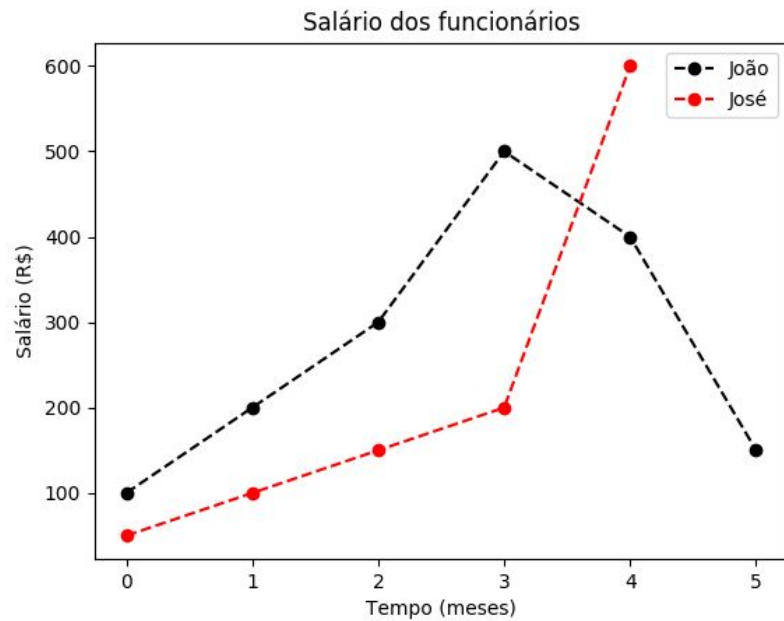
```
>>> plt.title('Salários dos funcionários')
```

```
>>> plt.savefig('salarios.png')
```

```
>>> plt.show()
```

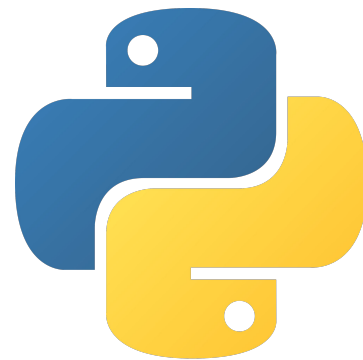


» MATPLOTLIB



» MATPLOTLIB

Para mais exemplos: <https://matplotlib.org/>



» PANDAS

- *pandas* é uma biblioteca *open source*, licenciada pela BSD, que fornece estruturas de dados de alto desempenho e fáceis de usar, além de ferramentas de análise de dados para a linguagem de programação Python (<https://pandas.pydata.org/>).
- <http://pandas.pydata.org/pandas-docs/stable/pandas.pdf>
- <https://www.cin.ufpe.br/~embat/Python%20for%20Data%20Analysis.pdf>



» PANDAS

- Mais alto nível que o *numpy*;
- Usada para dados tabulados;
- Utiliza o *numpy* por debaixo dos panos.

Instalação via pip:

- `sudo pip3 install pandas`

Instalação via aptitude:

- `sudo aptitude install python-pandas python-pandas-lib`



» PANDAS - SÉRIES

- Trabalhando com séries em pandas:

```
>>> import pandas as pd #convenção
```

```
>>> s = pd.Series([1,2,3,4,5,6,7,8,9])
```

```
>>> s.describe()
```

- Função describe retorna uma pequena descrição da série, com quantidade de elementos, média, desvio padrão, etc.



» PANDAS - SÉRIES

- Conta com diversas funções estatísticas:

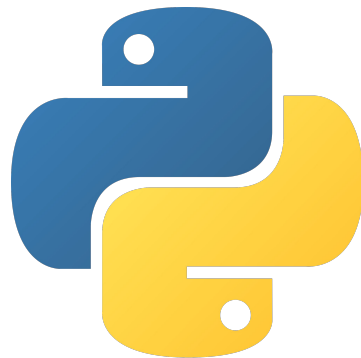
```
>>> s.mean()
```

```
>>> s.median()
```

- Concatenando uma série:

```
>>> s2 = pd.Series([10, 11, 12])
```

```
>>> s = s.append(s2)
```



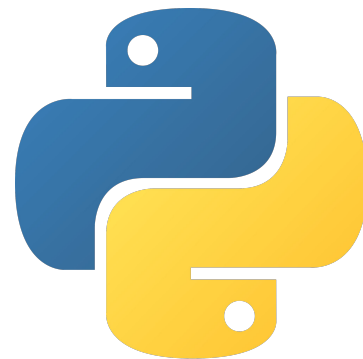
» PANDAS - DATAFRAMES

- DataFrames são estruturas de dados mais comumente utilizadas.
(Formato de matriz)

```
>>> import pandas as pd
```

```
>>> df = pd.DataFrame([[ 'info1', 1], [ 'info2', 2], [ 'info3', 3]])
```

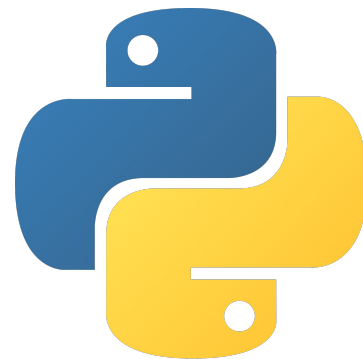
```
>>> print(df)
```



» PANDAS - DATAFRAMES

- Notem que os dados são tabulados, mas as linhas e colunas têm informações numéricas, que não fazem sentido para nós.
- Melhorando:

```
>>> df = pd.DataFrame([[ 'info1', 1], [ 'info2', 2], [ 'info3', 3]],  
columns=[ 'Informacao', 'quantidade'])  
  
>>> print(df)
```



» PANDAS - DATAFRAMES

- Acesso às informações no DataFrame;

```
>>> df[0] #Erro, não há coluna 0. (Existem as colunas 'Informacao' e  
'quantidade')
```

Correto:

```
>>> df['Informacao']
```

```
>>> df['quantidade']
```



» PANDAS - DATAFRAMES

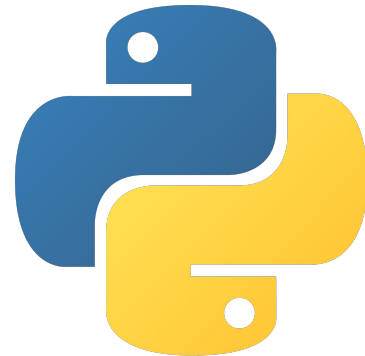
- Note que as operações retornaram séries, então, DataFrames são conjuntos de séries !!
 - Para concatenar uma série ao dataframe, é igual criar uma nova posição em um dicionário !

Acessando uma linha completa:

- Método `iloc[valor]`;
- Retorna um dicionário;

```
>>> df.iloc[0]
```

```
>>> df.iloc[0]['quantidade']
```



» PANDAS - DATAFRAMES

- Acessando a partir de índice:
- `pandas.DataFrame.ix[index]`

```
>>> df.ix[0]
```

Retorna todas as informações do **ÍNDICE** 0, não necessariamente a posição 0.

- ix = Índice.
- iloc = Posição



» PANDAS - DATAFRAMES

- Mudando índices:

```
>>> df.index = pd.Index([3,2,1])
```

```
>>> print(df)
```

- Notem que o índice do dataframe agora é outra coisa, ou seja, ix e iloc podem ter valores diferentes!



» PANDAS - DATAFRAMES

- Lendo datasets com pandas.
 - Iremos utilizar o arquivo 'copacabana.csv' e utilizaremos o método *read_csv()*.
 - Será demonstrado apenas com csv, para outros formatos, consulte a documentação.

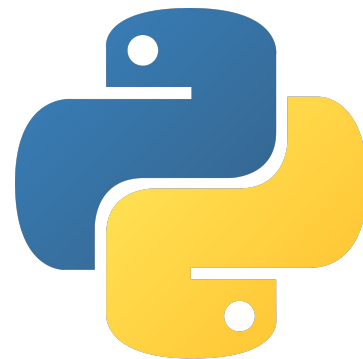


» PANDAS - DATAFRAMES

```
>>> import pandas as pd
```

```
>>> copacabana = pd.read_csv('copacabana.csv', delimiter=';')
```

- Pronto, carregamos um dataset enorme em 2 linhas. É nós, abraço, valeu!!



» PANDAS - DATAFRAMES

- Considerações importantes sobre o método `read_csv()`, o método foi implementado de forma que a primeira linha do seu arquivo csv seja o cabeçalho do arquivo, se você fizer um print no dataframe, irá perceber que as colunas já estão renomeadas de acordo com o que foi lido no cabeçalho.
- O delimitador padrão americano é a vírgula, como nós comumente usamos o ponto e vírgula, é preciso mudar.



» PANDAS - DATAFRAMES

- Filtragem e seleção.
 - Vamos dar uma olhada no dataset de copacabana!

```
>>> copacabana['Quartos'].describe()
```

- Note que o valor mínimo não é 0 por tal dataset sofrer testes estatísticos, e que o valor máximo é 6. (Tais valores representam quantidade de quartos de imóveis em copacabana)



» PANDAS - DATAFRAMES

- Então, se eu quiser saber quais são os quartos com 6 quartos?

```
>>> copacabana['Quartos'] > 5
```

- Vai criar uma série booleana de correspondentes à série do dataframe.



» PANDAS - DATAFRAMES

- Localizando um elemento:
 - método `loc()`;
 - Permite busca por intervalo. (Indexação por *B-pluss Tree*)

```
>>> copacabana.loc[copacabana['Quartos'] == 6]
```

```
>>> copacabana.loc[copacabana['Quartos'] >= 5] #para 5 e 6 quartos
```



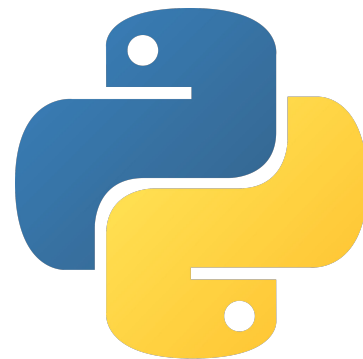
» PANDAS - DATAFRAMES

- Categorização de dados
 - Usado para aumentar performance no *dataset*;
 - Só faz sentido quando os dados são qualitativos.
 - Trabalharemos com o *dataset* 'titanic'
- Arquivo 'categorizacao.py'



» PANDAS - DATAFRAMES

- Os arquivos *matplot.py* e *disperc.py* apresentam exemplos de utilização do *matplotlib.pyplot* diretamente pelo *pandas*.
- Deixo-os como curiosidade !!



» NLTK

- *Natural Language Toolkit*
- NLTK é uma plataforma de programação python para processar dados em linguagem humana.
- Possui módulos para classificação, *tokenização*, *stemming*, *tagging*, análise e raciocínio semântico, *wrappers*, etc.
- <https://www.nltk.org/>



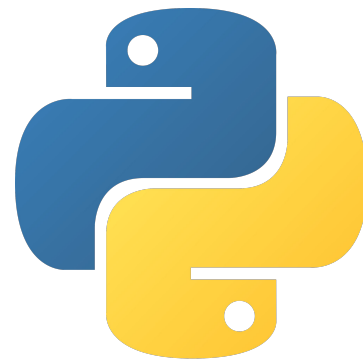
» NLTK

- Será demonstrado o uso do NLTK para pré-processamento de dados de texto.
- A mesma pode ser usado para outros fins também!



» NLTK - STOPWORDS

- *Stopwords* são palavras consideradas não relevantes para a análise de texto, justamente por não traduzirem sua essência. Normalmente fazem parte dessa lista as preposições, pronomes, artigos, advérbios e outras palavras auxiliares.



» NLTK - STOPWORDS

- Para remoção de *stopwords*, o NLTK conta com o pacote *stopwords* que está no módulo *corpus*. Lá existem stopwords da maioria das línguas.

```
>>> import nltk
```

```
>>> stpw = nltk.corpus.stopwords.words('english')
```

```
>>> print(stpw)
```



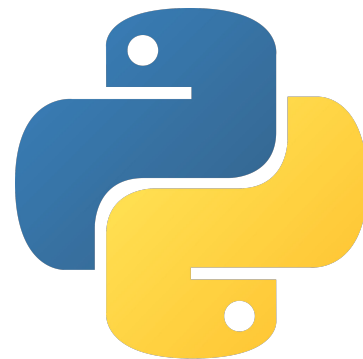
» NLTK - STOPWORDS

- Com o vetor de *stopwords*, basta fazer uma simples função que vai *tokenizar* seu conjunto de palavras e remover as que estejam no vetor de *stopwords*.
- O arquivo `script.py` da pasta NLTK demonstra isso a partir da função `remove_words()`



» NLTK - STEMMING

- Durante processos de indexação, dependendo do caso, é interessante retirar as variações morfológicas de uma palavra. Tais variações são retiradas identificando o radical da palavra. Assim, prefixos e sufixos são retirados e apenas o radical da palavra permanece.



» NLTK - STEMMING

- A NLTK oferece o pacote *stem* que contém diversos módulos para fazer *stemming* de texto.
- O pacote para *stemming* específico de português é o *RSLPStemmer*.
- Para inglês, o mais usado é o *SnowballStemmer*, que conta com diversas linguagens e deve ser explícita no momento de instanciação.



» NLTK - STEMMING

```
>>> stm = nltk.stem.RSLPStemmer() #nao especifica, unico de port.
```

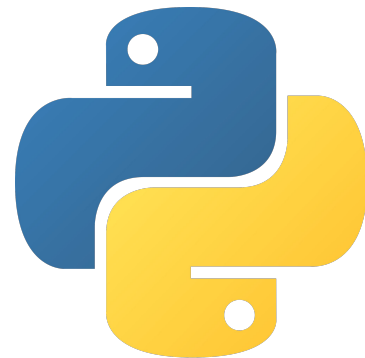
```
>>> stm2 = nltk.stem.SnowballStemmer('english')
```

- Envolve o mesmo processo de remoção de *stopwords*,
 - Tokens
 - Aplicar *stm.stem(token)*
- A função *apply_stemmer()* demonstra o processo.



» SCIKIT LEARN

- Biblioteca para *Machine Learning* em Python.
 - Ferramentas simples para mineração e análise de dados;
 - Construído em Numpy, SciPy e matplotlib
 - Open source
 - <http://scikit-learn.org/>



» SCIKIT LEARN - FEATURE EXTRACTION

- Módulo *Feature Extraction* trata de extração de características de diversas formas de dado.
 - Será demonstrado por Texto.



» SCIKIT LEARN - FEATURE EXTRACTION

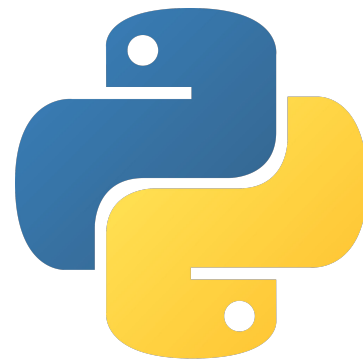
- TFIDF - *Term Frequency-Inverse Document Frequency*
 - Utilizado para dar peso para as palavras de um documento, trazendo sua relevância em conta.

$$\text{Peso}_{td} = \text{Freq}_{td} / \text{DocFreq}_{td}$$



» SCIKIT LEARN - FEATURE EXTRACTION

- Arquivo *tfidf.py* demonstra a utilização do módulo



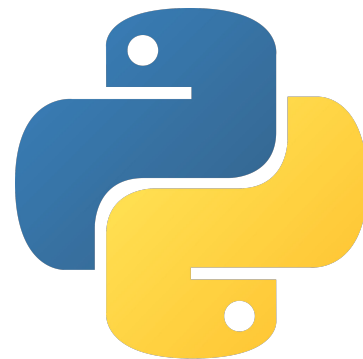
» SCIKIT LEARN

- Breve explicação sobre *Machine Learning*
 - Aprendizado Supervisionado
 - Aprendizado Não Supervisionado
 - Aprendizado por Reforço



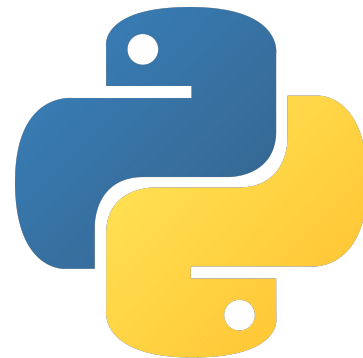
» SCIKIT LEARN

- Aprendizado Supervisionado
 - Leva em conta outras observações, ou seja, há um treinamento com amostras da base de dados;
 - O sistema recebe a resposta correta durante o treino;
 - Cria uma função para mapear as entradas para a saída;
 - Eficiente;
 - Simples em relação aos outros;



» SCIKIT LEARN

- Aprendizado Não Supervisionado
 - Deve reconhecer padrões nas amostras sem feedback de saída;
 - Complexo.



» SCIKIT LEARN

- Aprendizado por Reforço
 - Sistema de recompensas e punições;
 - O sistema aprende por várias observações.



» SCIKIT LEARN - SUPPORT VECTOR MACHINES

- Iremos tratar de um modelo de aprendizado supervisionado chamado *Support Vector Machine*.
 - Classificador linear binário não probabilístico;
 - Cria um hiperplano entre as classes;
 - Busca maximizar a distâncias entre os pontos e as margens do plano;

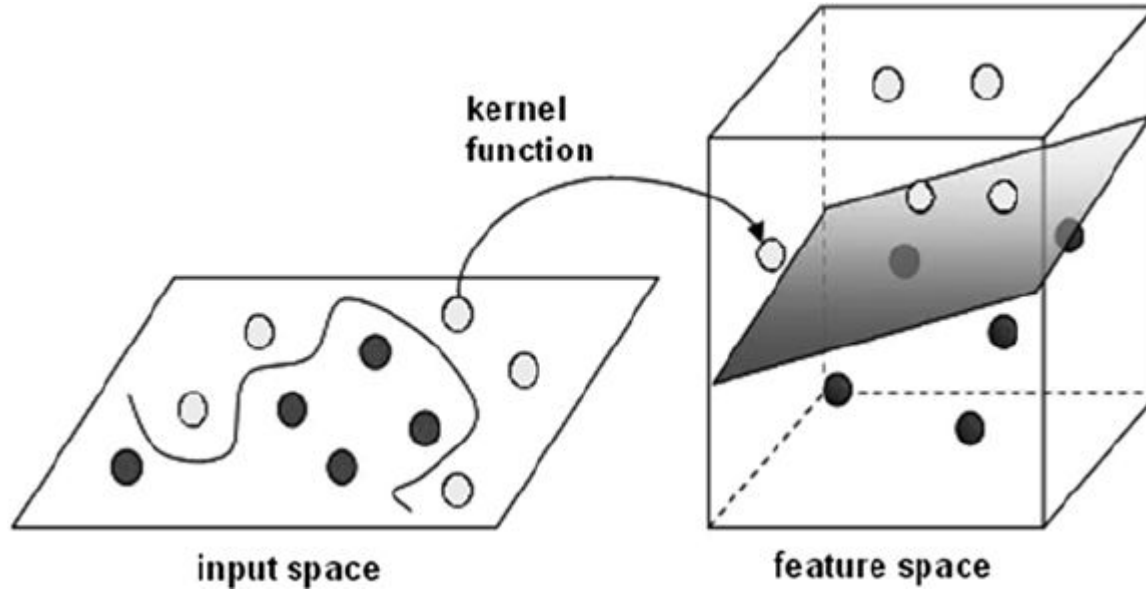


» SCIKIT LEARN - SUPPORT VECTOR MACHINES

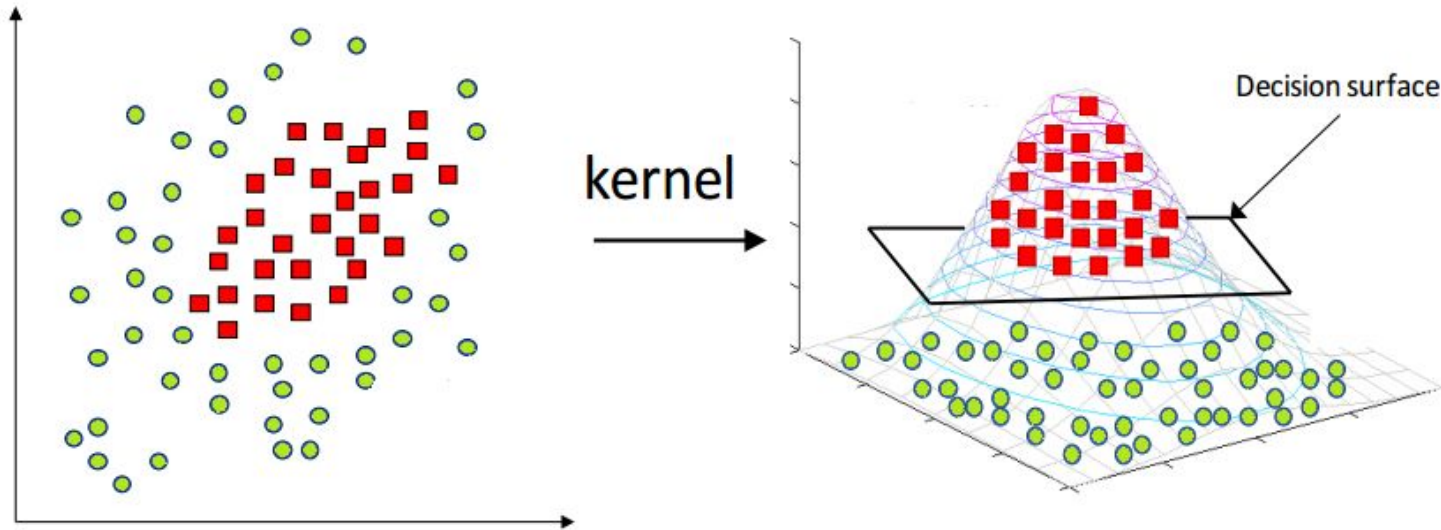
- Para maximizar as distâncias entre os pontos e as margens do plano, é utilizado uma função de *kernel* para trabalhar os dados de forma que os mesmos consigam sofrer separação por dimensão.



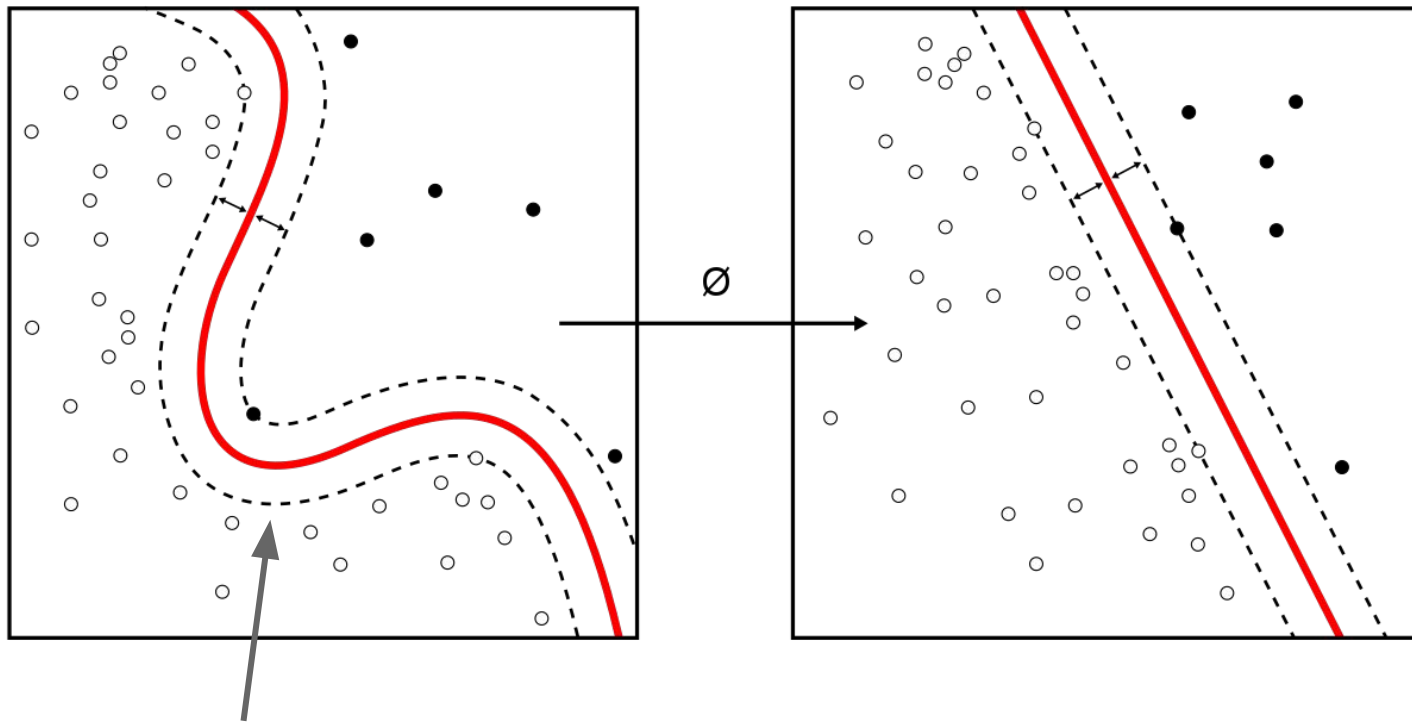
» SCIKIT LEARN - SUPPORT VECTOR MACHINES



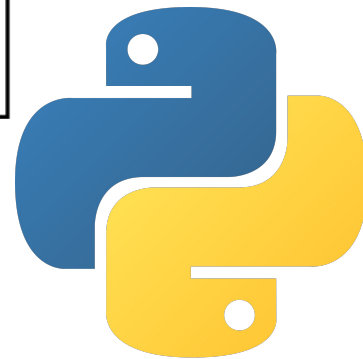
» SCIKIT LEARN - SUPPORT VECTOR MACHINES



» SCIKIT LEARN - SUPPORT VECTOR MACHINES

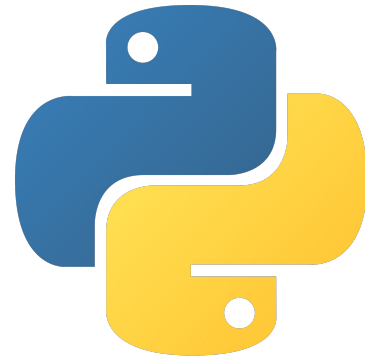


Margem



» SCIKIT LEARN - SUPPORT VECTOR MACHINES

- O arquivo `svm.py` demonstra a utilização do *sklearn* para fazer classificação dos dados do Iris dataset.



» SCIKIT LEARN - CROSS VALIDATION

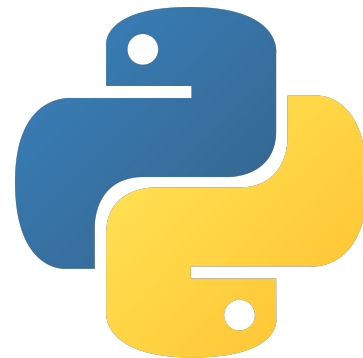
- Para validar um modelo de classificação existem diversas técnicas. A mais famosa é o *cross validation* por n camadas.
 - Vai dividir o *dataset* em n partes de treino e teste diferentes;
 - Classificar todos os n novos *datasets* de acordo com a divisão;
 - Tirar a média dos resultados obtidos.
- O arquivo *crossvalidator.py* demonstra tal método.



DÚVIDAS ??

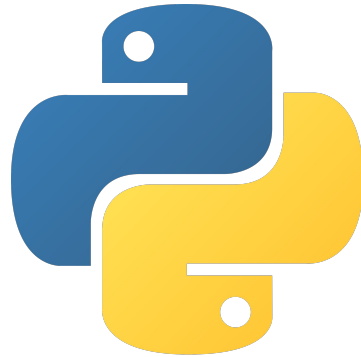
O Diretório Acadêmico da Engenharia de Computação agradece a sua presença !

HACKING WITH PYTHON !!!



ORM E WEB

Banco de dados & Web de forma fácil

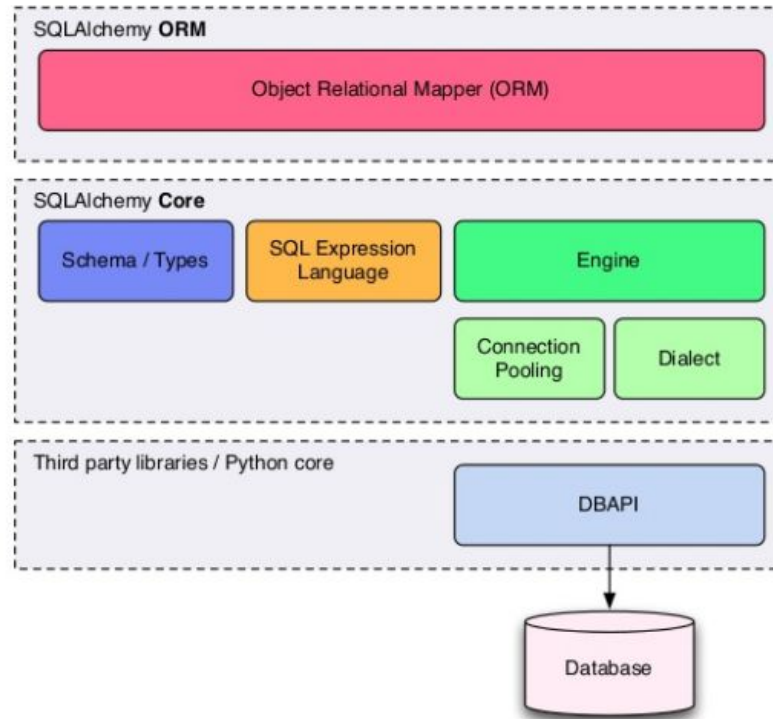


SQLALCHEMY

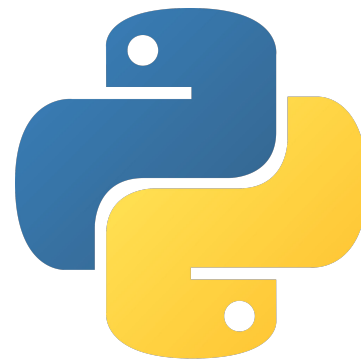
- **O**bject **R**elational **M**apper
 - Classes <-> Tabelas
 - Objetos <-> Registros
- DDL & DML baseado em funções & métodos
- Eager Loading
- Transparente em relação ao banco de dados utilizado
 - Conceito de **Dialect**
 - <http://docs.sqlalchemy.org/en/latest/dialects/>



ARQUITETURA DO SQLALCHEMY



CRIANDO UM BANCO DE DADOS MODELO



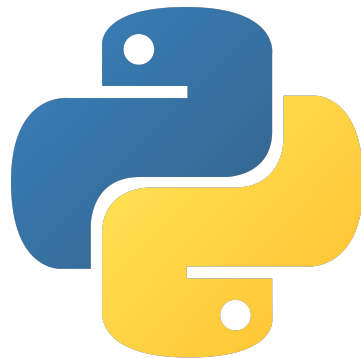
DB.PY

```
from sqlalchemy import *  
  
from sqlalchemy.orm import *  
  
from sqlalchemy.ext.declarative import *  
  
# metadata da modelagem  
  
Base = declarative_base()
```



DB.PY

```
class Categoria(Base):  
    __tablename__='categorias'  
    id = Column(Integer,primary_key=True)  
    nome = Column(String(500))  
    produtos = relationship('Produto')
```



DB.PY

```
class Produto(Base):  
    __tablename__='produtos'  
  
    id = Column(Integer,primary_key=True)  
  
    nome = Column(String(500))  
  
    preco = Column(Numeric(precision=10,scale=2))  
  
    categoria_id = Column(Integer,  
ForeignKey('categorias.id'))  
  
    categoria = relationship('Categoria')
```



DB.PY

```
assoc_produtos_vendas = Table('produtos_vendas', Base.metadata,  
    Column('produto_id', Integer, ForeignKey('produtos.id')),  
    Column('venda_id', Integer, ForeignKey('vendas.id'))  
)
```



DB.PY

```
class Venda(Base):  
    __tablename__='vendas'  
  
    id = Column(Integer,primary_key=True)  
  
    data = Column(DateTime)  
  
    produtos = relationship('Produto',  
                             secondary=assoc_produtos_vendas)
```



DB.PY

```
# Cria a conexão com o banco de dados e as tabelas
```

```
engine =  
create_engine('mysql://root:root@localhost/curso_python',  
echo=True)
```

```
dbSession = sessionmaker()
```

```
dbSession.configure(bind=engine)
```

```
Base.metadata.create_all(engine)
```



INSERTS.PY

```
from db import *
```

```
from datetime import *
```

```
# Cria as categorias
```

```
c1 = Categoria(nome='Massas')
```

```
c2 = Categoria(nome='Limpeza')
```



INSERTS.PY

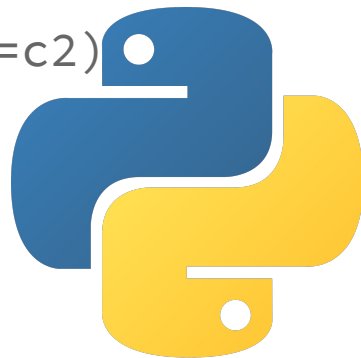
```
# Cria produtos
```

```
p1 = Produto(nome='Macarrão',preco=9.9,categoria=c1)
```

```
p2 = Produto(nome='Canelone',preco=15.9,categoria=c1)
```

```
p3 = Produto(nome='Sabão em pó',preco=14.9,categoria=c2)
```

```
p4 = Produto(nome='Detergente',preco=5.8,categoria=c2)
```



INSERTS.PY

```
# Vendas
```

```
v1 = Venda(produtos=[p1,p2,p4],data=datetime(2018,9,10,15,0))
```

```
v2 = Venda(produtos=[p2,p3,p1],data=datetime(2018,9,1,21,0))
```

```
v3 = Venda(produtos=[p1],data=datetime(2018,6,1,8,45))
```



INSERTS.PY

```
# Envia as transações para o banco
```

```
sess = dbSession()
```

```
sess.add_all([c1,c2,p1,p2,p3,p4,v1,v2,v3])
```

```
sess.commit()
```



CONSULTAS

- A partir da sessão podemos consultar as entidades do banco
- O objeto query armazena a estrutura da consulta
 - **get** → Recupera um registro pela chave primária
 - **filter_by** → Estrutura da condição Where
 - **order_by** → Cláusula ORDER BY
 - **group_by** → Cláusula GROUP BY
 - **having** → Cláusula HAVING
- Retornar os registros
 - **first** → Primeiro registro
 - **all** → Todos (lista)
 -



SELECTS.PY

```
# Session
```

```
sess = dbSession()
```

```
# Categoria de ID=1
```

```
print(sess.query(Categoria).get(1))
```



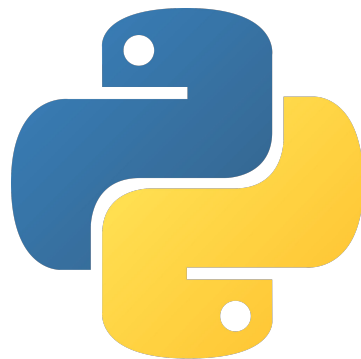
SELECTS.PY

```
# Produto com o nome canelone
```

```
print(sess.query(Produto).filter_by(nome =  
'Canelone').all())
```

```
# A categoria do produto Macarrão
```

```
print(sess.query(Produto)  
.filter_by(nome='Macarrão').first().categoria)
```



SELECTS.PY

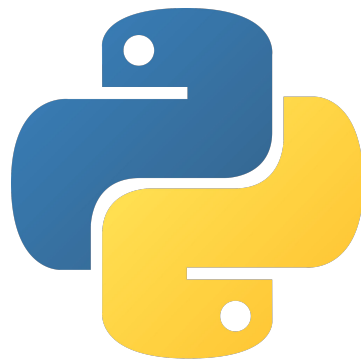
```
# Produtos da venda ID=3
```

```
print(sess.query(Venda).get(3).produtos)
```



FLASK

- Micro framework
- Objetiva reutilizar código
- Documentação extensa
- Ideal para projetos pequenos



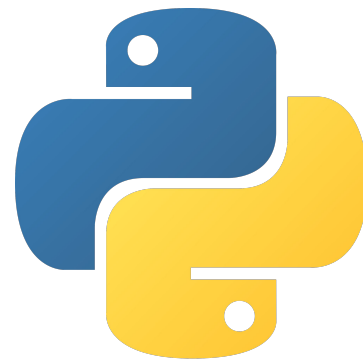
FLASK

- Cookie-based session
- Flashing
- Mapeamento RESTful
- Hooks antes/depois do processamento da requisição

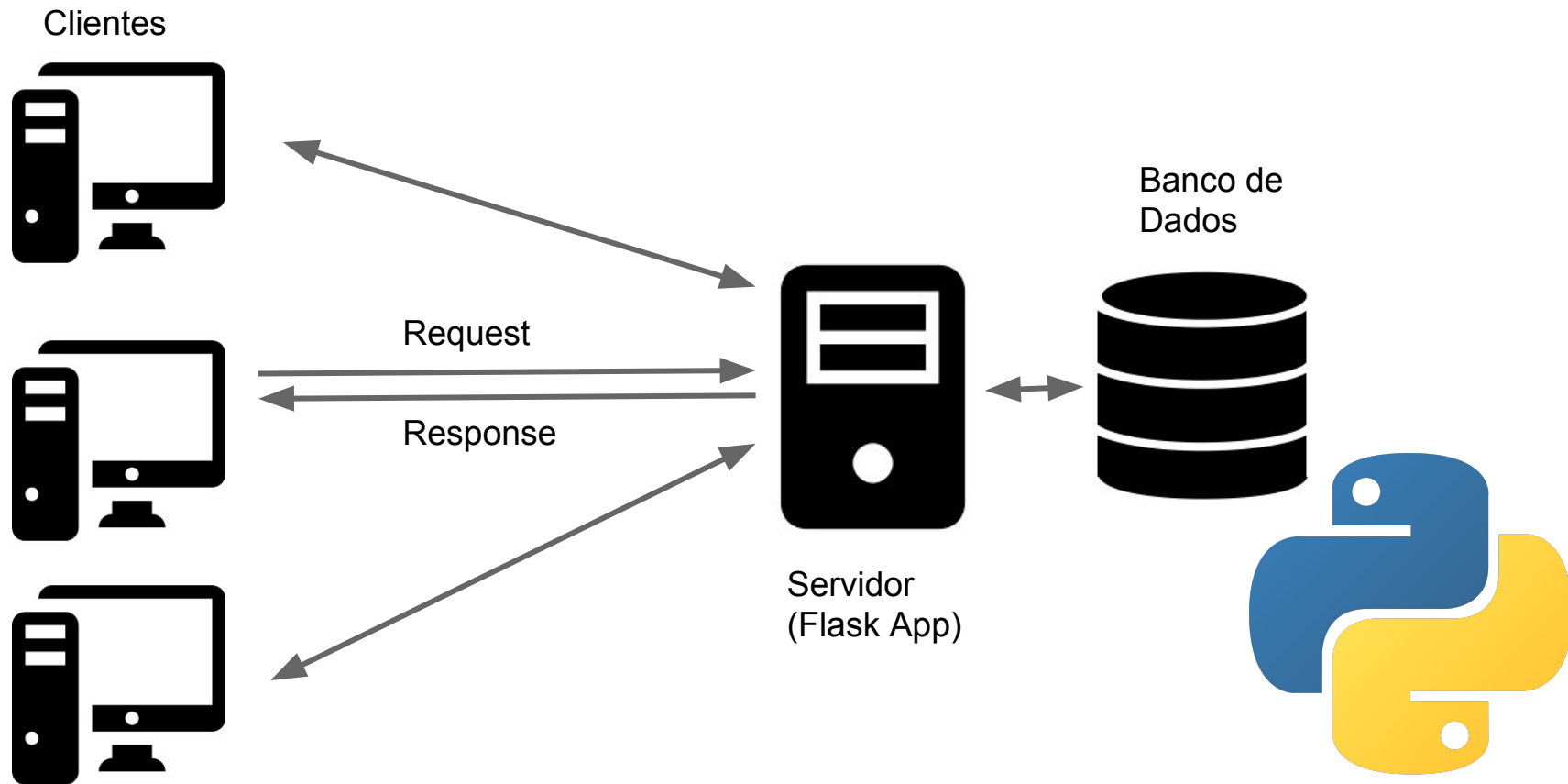


FLASK

- XML-RPC
- API de Registro (OAuth, OpenID)
- Segurança (CSRF protection)



FLASK



HELLOWORLD_FLASK.PY

```
from flask import *  
app = Flask(__name__)  
  
@app.route('/')  
def home():  
    return 'Hello world flask!'  
  
if __name__ == '__main__':  
    app.run(debug=True)
```



JINJA2

- Template Engine
- Constrói blocos de comandos no HTML que serão processados
- Suporta filtros e template extensions
- Variáveis declaradas são enviadas para o template



ORGANIZAÇÃO DO PROJETO

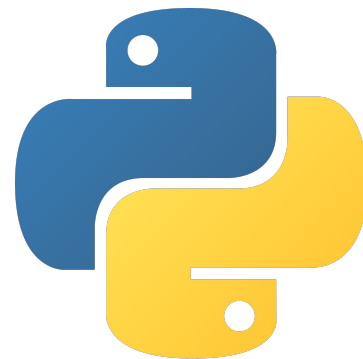
- **app.py** → Código da aplicação
- **static** → Arquivos estáticos do projeto (IMG, CSS, JS...)
- **templates** → Arquivos do Jinja2 template
 - Extensão dos templates é “html”



CONFIGURAÇÃO DO APP

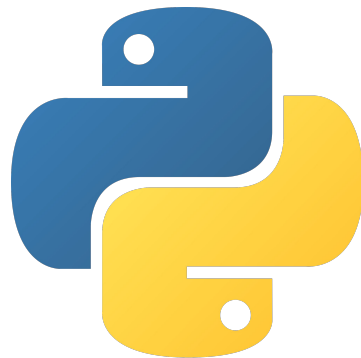
```
from db import *  
from flask import *
```

```
app = Flask(__name__)  
app.secret_key = 'CuRS0$PyTH0N'
```



LISTAR CATEGORIAS E VENDAS (APP.PY)

```
@app.route('/')  
def index():  
    sess = dbSession()  
    return render_template('index.html',  
        categorias=sess.query(Categoria).all(),  
        vendas=sess.query(Venda).all()  
    )
```



LISTAR CATEGORIAS E VENDAS (INDEX.HTML)

```
<html>
  <head>
    <title>Controle de vendas</title>
  </head>
  <body>
    {% with msg = get_flashed_messages() %}
      {% for m in msg %}
        <h1><font color="red">{{ m }}</font></h1>
      {% endfor %}
    {% endwith %}
```



LISTAR CATEGORIAS E VENDA (INDEX.HTML)

```
<h1>Categorias cadastradas</h1>
```

```
<a href="{{ url_for('nova_categoria') }}">Nova categoria</a>
```

```
<ul>
```

```
    {% for c in categorias %}
```

```
        <li>{{ c.nome }}</li>
```

```
    {% endfor %}
```

```
</ul>
```



LISTAR CATEGORIAS E VENDAS (INDEX.HTML)

```
<h1> Vendas cadastradas</h1>
<ul>
  {% for v in vendas %}
    <li>{{ v.data }}
    <a href="{% url_for('excluir_venda',id=v.id) %}">Excluir</a>
    <ol>
      {% for p in v.produtos %}
        <li>{{ p.nome }} {{ p.preco }}</li>
      {% endfor %}
    </ol>
  </li>
{% endfor %}
</ul>
```



CADASTRAR CATEGORIA (APP.PY)

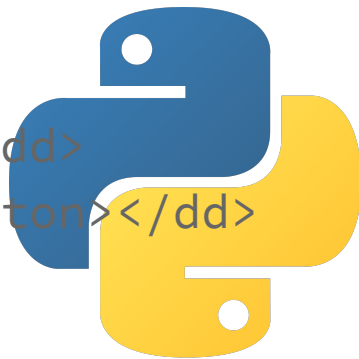
```
@app.route('/nova_categoria', methods=['GET', 'POST'])
def nova_categoria():
    sess = dbSession()
    if request.method == 'POST':
        c = Categoria(nome=request.form['nome'])
        sess.add(c)
        sess.commit()
        flash('Categoria criada com sucesso!')
        return redirect('/')

    return render_template('nova_categoria.html')
```



CADASTRAR CATEGORIA (NOVA_CATEGORIA.HTML)

```
<html>
  <head>
    <title>Controle de vendas</title>
  </head>
  <body>
    <form method="POST">
      <dl>
        <dt>Nome:</dt>
        <dd><input type="text" name="nome"></dd>
        <dd><button type="submit">Salvar</button></dd>
      </dl>
    </form>
```



EXCLUIR VENDA

```
@app.route('/excluir_venda')
def excluir_venda():
    sess = dbSession()
    v = sess.query(Venda).get(request.args['id'])
    sess.delete(v)
    sess.commit()
    flash('Venda excluída com sucesso!')

    return redirect('/')
```



AGORA É COM VOCÊ!!!



- **Construa as telas de editar**
 - Use um **input hidden** para salvar o ID e use o método update para alterar o registro
 - Construa as demais telas
 - Tente colocar o bootstrap na APP (não é difícil!!!)



MUITO OBRIGADO!!!!

