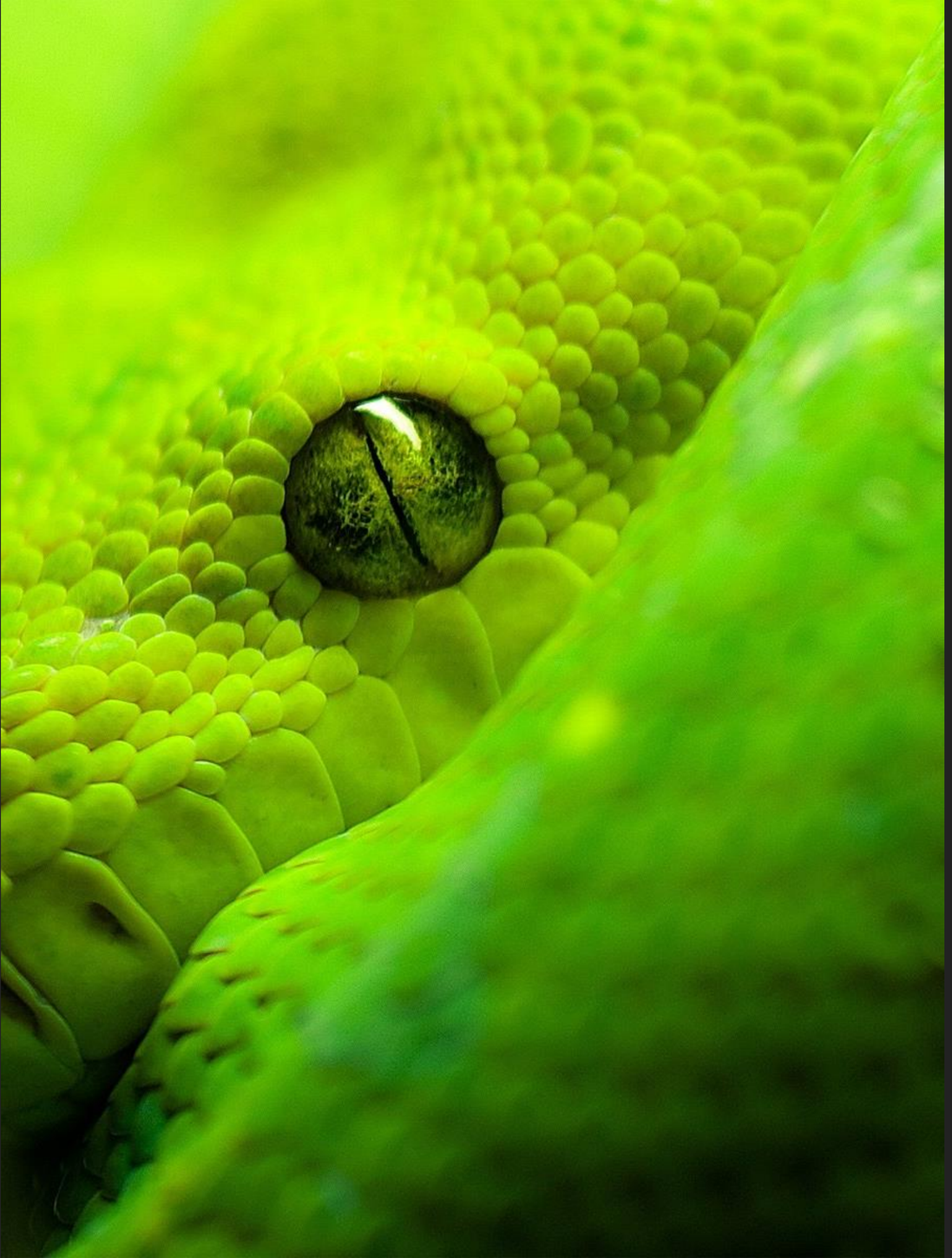
PROGRAMAÇÃO PARA COMPUTADORES

PYTHON

@AlexFurtunato

- QUALIDADE DE SOFTWARE
- PRODUTIVIDADE
- PORTABILIDADE
- BIBLIOTECAS DISPONÍVEIS
- INTEGRAÇÃO DE COMPONENTE
- DIVERTIDA

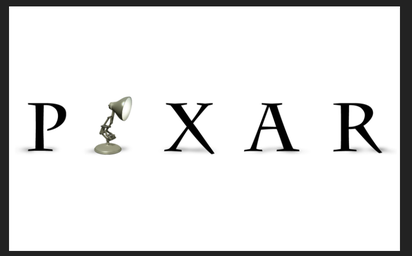
POR QUE AS PESSOAS USAM PYTHON?

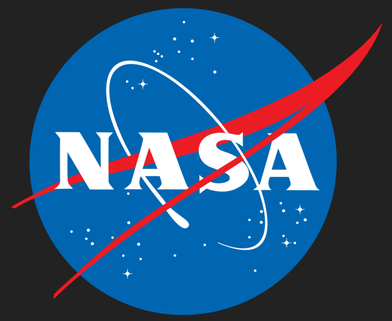


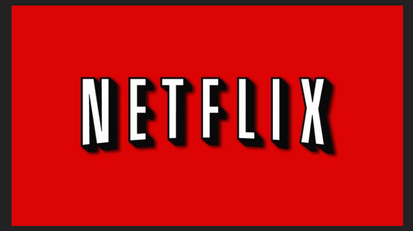
- É ORIENTADA A OBJETOS E FUNCIONAL
- É LIVRE
- TEM UMA VASTA COMUNIDADE ATUANTE
- É PORTÁVEL
- É PODEROSA
- É “MISTURÁVEL" COM OUTRAS LINGUAGENS
- É FÁCIL DE USAR
- É FÁCIL DE APRENDER

QUAIS AS FORÇAS DO PYTHON?

QUEM USA O PYTHON?







O QUE POSSO FAZER COM PYTHON?

- Programação de sistemas - Scripts
- GUIs - Tkinter ou wxPython, PyQT, PyGTK, PyWin32
- Internet Scripting - Sockets, JSON, XML-RPC, Django
- Integração - C e C++, Cython, Jython, IronPython
- Banco de Dados - ODBC, MySQL, PostgreSQL, SQLite
- Rápida Prototipagem - Componentes a serem compilados

- Programação Científica e Numérica - NumPy, SciPy
- Games - PyGame, Pyglet, PySoy, Panda3D
- Processamento de Imagens - PIL, PyOpenGL, Blender, Maya
- Robôs - PyRo
- Instrumentação - Raspberry PI, Arduino

VERSÕES DO PYTHON

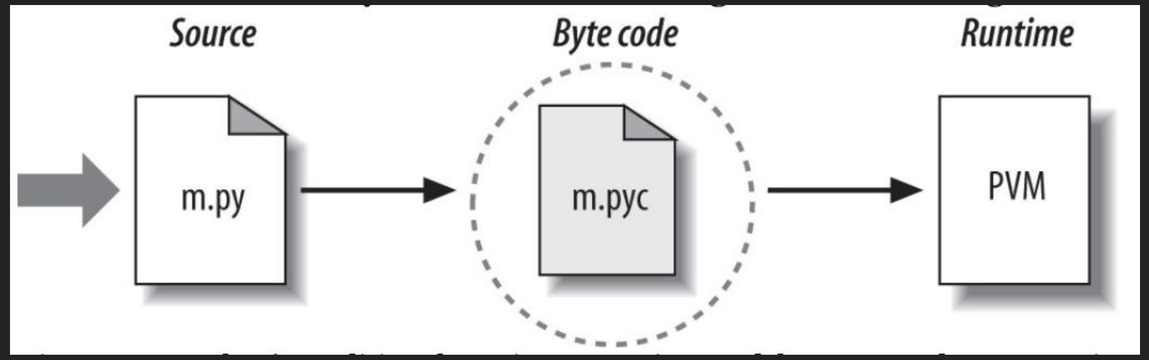
- Python 2.x é considerada Legado
  - 2.7 lançada no final de 2010
  - Utilizada principalmente por questões de compatibilidade com bibliotecas
  - Não deve haver mais lançamentos de versões dentro da família 2.x
  - As distribuições Linux e o MacOS ainda mantém essa versão como oficial no S.O.

- Python 3.x é a versão estável
  - Python 3.0 foi lançado em 2008. Python 3.5 foi liberado em 2015
  - Preferencialmente deve ser a versão a ser escolhida
  - O desenvolvimento futuro será apenas na familia 3.x
  - Existe um esforço da comunidade para portar as bibliotecas para a versão 3.x

INSTALAÇÃO DO PYTHON

- Linux e MaxOS
  - Já vem instalado por padrão
  - Basta abrir o terminal e iniciar o uso
- Windows
  - Baixar em http://www.python.org/
  - Executar o instalador NNF (Next, Next, Finish)
  - Já inclui um terminal Shell para comandos e uma IDE

EXECUÇÃO DE UM CÓDIGO PYTHON



- Código fonte com extensão .py
- O Interpretador gera um Byte Code Python .pyc
  - Automaticamente
- O Python Virtual Machine (PVM) executa o Byte Code
  - Próximas execuções utilizarão o Byte Code gerado para tornar a execução mais rápida
  - Monitora se houve mudança do Código Fonte a cada execução

COMO EXECUTAR CÓDIGO EM PYTHON

- Prompt Interativo

```
 Python 3.5.1 (v3.5.1:37a07cee5969, Dec  6 2015, 01:38:48) [MSC v.1900
32 bit (Intel)] on win32

 Type "copyright", "credits" or "license()" for more information.

 >>>
```

  - Interpretador instalado como um programa executável
  - Utilizado para testes de código e experimentações
  - No Windows, Basta chamar Python.exe (Ou basta chamar "py" do prompt). No Linux e MacOS, chama o comando python pelo Terminal Shell

EXEMPLO DE CÓDIGO

```
 Python 3.5.1 (v3.5.1:37a07cee5969, Dec  6 2015, 01:38:48) [MSC v.1900 32
bit (Intel)] on win32

 Type "copyright", "credits" or "license()" for more information.

 >>> print(“Hello World!”)

 Hellow World!

 >>> print(2**8)

 256

 >>> mensagem=“okay"

 >>> print(mensagem)

 okay
```

EXEMPLO DE CÓDIGO

```
 Python 3.5.1 (v3.5.1:37a07cee5969, Dec  6 2015, 01:38:48) [MSC
v.1900 32 bit (Intel)] on win32

 Type "copyright", "credits" or "license()" for more information.

 >>> X

 Traceback (most recent call last):

   File "<pyshell#3>", line 1, in <module>

     X

 NameError: name 'X' is not defined

 >>>
```

COMO EXECUTAR CÓDIGO EM PYTHON

- Módulo (Programa ou Script Python)
  - Módulo são códigos python salvos arquivos
  - Os arquivos são salvos com extensão .py
  - Módulos podem ser importados para outros arquivos
  - Pode ser utilizado qualquer editor de texto (vi, Notepad ou IDEs)

EXEMPLO DE CÓDIGO

```
# O primeiro script python

import sys              # Importa um módulo (biblioteca adicional)

print(sys.platform)     # Imprime a versão da plataforma S.O.

print(2**100)           # Imprime o resultado de dois elevado a cem

x = 'Spam!'             # Cria uma variável e armazena uma string

print(x*8)              # Imprime a string oito vezes


# Salve esse código num arquivo com nome "script1.py"
```

EXECUÇÃO DE MÓDULO

No Windows

```
C:> py script1.py
```

No Linux ou MacOs:

```
maquina: ~ usuario$ python script1.py
```

UNIX STYLE EXECUTION SCRIPT

- Para tornar um script python como executável nos sistemas Unix-Like (Unix, Linux, MacOs)
  - Na primeira linha do Módulo adiciona-se um código especial (Com o caminho do interpretador python):

```
#!/usr/local/python
```

  - Adiciona-se privilégios de execução ao arquivo:

```
chmod +x arquivo.py
```

- O módulo agora pode ser executado diretamente no prompt

```
maquina: ~usuario$ arquivo.py
```

PYTHON E A INDENTAÇÃO DE CÓDIGO

- Em Python não existe delimitadores de blocos de código.
- Python usa o recurso de indentação de código para marcar blocos de código

```
x = 1

if x == 1:

    y = 2

    print(y)

print(x)
```

- Por recomendação da PEP8 (Python Enhancement Proposal - Proposta de Aprimoramento do Python) deve-se usar 4 (quatro) espaços por nível de indentação
- Não se deve misturar espaços e tabulações em indentação
- Os Editores e IDEs para Python já utilizam esse formato de indentação

INDENTAÇÃO

<table>
<tr><td colspan="3">BLOCO 0</td></tr>
<tr><td colspan="3">LINHA DE CABEÇALHO:</td></tr>
<tr><td>4 ESPAÇOS</td><td colspan="2">BLOCO 1</td></tr>
<tr><td>4 ESPAÇOS</td><td colspan="2">LINHA DE CABEÇALHO:</td></tr>
<tr><td>4 ESPAÇOS</td><td>4 ESPAÇOS</td><td>BLOCO 2</td></tr>
<tr><td>4 ESPAÇOS</td><td colspan="2">BLOCO 1</td></tr>
<tr><td colspan="3">BLOCO 0</td></tr>
</table>

- A indentação tem uma vantagem extra:

  “Os código em Python seguem uma organização padrão”

TUDO É OBJETO EM PYTHON. MESMO UM SIMPLES NÚMERO, UM MÓDULO, UMA FUNÇÃO

HIERARQUIA CONCEITUAL DO PYTHON

1. *Programas* são compostos por *Módulos*
2. *Módulos* contém *Declarações*
3. *Declarações* contém *Expressões*
4. *Expressões* criam ou processam *Objetos*

IMPORTÂNCIA DOS OBJETOS “BUILT-IN” DO PYTHON

- Objetos *“Built-in”* tornam os programas mais fáceis
- Objetos *“Built-in”* são componentes de extensões
- Objetos *“Built-in” são mais eficientes que estruturas customizadas*
- *Objetos “Built-in” são parte padrão da linguagem*

OBJETOS “BUILT-IN” DO PYTHON

Tipo	Exemplo
Números	1234, 3.1415, Decimal()
Strings	Spam’, “Bob’s"
Listas	[1,[2, ‘tres’], 4, 5], list(range(10))
Dicionários	{‘comida’:’spam’, ‘cor’: ‘azul’}
Tuplas	(1, ‘spam’, 4, ‘U’)
Arquivos	open(‘arquivo.txt’)
Sets	set(‘abc’), {‘a’, ‘b’, ‘c’}
Outros tipos	Boolean, types, None

LISTAR ATRIBUTOS DE UM OBJETO

- Para listar os atributos de um objeto em python, utilize o método ‘dir()’:

```
>>> x = ‘spam’

>>> dir(x)

 ['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__',
'__hash__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold',
'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map',
'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
```

EXEMPLOS DE USO DE OBJETOS BUILT-IN

```
>>> S = ‘spam'
>>> len(S)
>>> S[0]
>>> S[1:3]
>>> S + ‘xyz'
>>> S.find(‘pa')
>>> S.upper()
>>> S.isalpha()
>>> S.replace(‘pa’,'ca')
```

```
>>> L = [123, ‘casa’, 3.14]
>>> len(L)
>>> L[0]
>>> L * 2
>>> L.append(‘bola’)
>>> M = [[1,2,3],[4,5,6],[7,8,9]]
>>> col2 = [n[1] for n in M]
>>> D = {‘com’:’ABC', ‘num’:23}
>>> D[‘num’]
>>> D[‘cor’] = ’azul'
```

NÚMEROS

- Inteiros

  `123, -25, 0`

- Ponto flutuante

  `3.14, 4e12`

- Octal, Hexadecimal, binário

  `0o177, 0x9ff, 0b1010`

- Complexo

  `3+4j, 3J`

- Boleano

  `True, False`

FERRAMENTAS NUMÉRICAS

- Operadores de expressões

```
+, -, *, /, //, **, %, >>, etc
```

- Funções matemáticas internas

```
pow, abs, round, int, hex, min, max, etc
 Ex: >>> min(3, 2, 1, 4)
 1
```

- Módulos utilitários

```
random, math, etc
 Ex: >>> math.sin(2 * math.pi / 180)
 0.03489949670250097
```

OPERADORES

- Precedência de operadores

```
a * b + c * d

# Consultar tabela de precedência em: https://docs.python.org/3/reference/

# /expressions.html?highlight=precedence#operator-precedence
```

- Agrupamento por parênteses

```
(x + y) * z   # Parênteses alteram precedência

x + (y * z)
```

- Mistura de Tipos

```
40 + 3.14   # Conversão para float antes da operação

40 + int(3.14) # Conversão explícita
```

OPERADORES DE COMPARAÇÃO

- Operadores que trabalham no nível de bits:

```
>>> x = 2
>>> y = 4
>>> z = 6
>>> x < y
True
>>> z < y
False
>>> x < y < z      # Equivalente a (x < y) and (y < z)
True
>>> x == z
False
```

NÚMEROS HEXADECIMAL E BINÁRIO

- O python pode fazer conversões rápidas entre Hexa, Binário e Decimal:

```
>>> 0xFF, 0b10000
(255, 16)
>>> hex(64)
'0x40'
>>> bin(64)
'0b1000000'
```

OPERADORES 'BITWISE'

- Operadores que trabalham no nível de bits:

```
>>> x = 1
>>> x << 2     # deslocamento dois bits a esquerda
4
>>> x | 2      # “Or" lógico bit-a-bit
3
>>> x & 1       # “And" lógico bit-a-bit
1
>>> y = 0b1000   # 8 em decimal
>>> y >> 3       # deslocamento de três bits a direita
1
```

FORMATAÇÃO DE NÚMEROS

- A formatação de um número, pode ser feito da seguinte forma:

```
>>> num = 1/3.0
>>> num
0.3333333333333333
>>> '%e' % num
'3.333333e-01'
>>> '%4.2f' % num
'0.33'
```

FUNÇÕES MATEMÁTICAS

- O Python tem módulos built-in com importantes funções matemáticas:

```
>>> import math
>>> math.sin(2*math.pi/180)   # trigonometria
>>> math.sqrt(144)            # raiz quadrada
>>> math.floor(2.543), math.trunc(2.543) # arredonda ou trunca
>>> import random
>>> random.random()     # gera um número randomico entre 0 e 1
>>> random.choice(['azul', 'amarelo', 'preto', 'branco'])
```

BOLEANOS

- Em Python, boleanos são valores numéricos cuja representação customizada é:
  - True (1) e False (0)

```
>>> type(True)
<class 'bool'>
>>> True == 1
True
>>> True and False
False
>>> True + 4
5
```

STRINGS

- String são objetos utilizados para representar qualquer coisa que possa ser codificada como texto ou sequencia de bytes
- Em Python, Strings são denominadas de “Sequências imutáveis”
  - Sequência porque trata-se de uma sequencia de caracteres
  - Imutável porque, uma vez criado, o objeto do tipo string não pode ser alterado
  - Ex:

```
>>> s = 'casa'

>>> x = ''
```

COTA SIMPLES(SIMPLE-QUOTED) OU COTA DUPLA (DOUBLE-QUOTED)

- Um objeto do tipo string pode ser criado com cota simples ou dupla. A diferença é apenas se dentro da string precisamos representar o caractere de cota simples (‘) ou de cota dupla (“)
  - Ex:

```
>>> w = “D’agua”

>>> z = ‘knight”s'
```

- É possível utilizar o mesmo tipo de cota, contudo deve-se usar o recurso de “Escapar” (Escape) os caracteres de cota dentro da string
  - Ex:

```
>>> w = ‘D\’agua'

>>> z = “knight\”s"
```

REPRESENTAÇÃO DE SEQUÊNCIAS DE ESCAPE

- O recurso de escapar caracteres serve também para incluir dentro de strings caracteres que não tem representação direta no teclado

```
>>> s = 'a\nb\nc'      # A sequencia \n significa Enter ou quebra de linha
>>> s
'a\nb\nc'
>>> print(s)
a
b
c
>>> len(s)
5
```

PRINCIPAIS SEQUÊNCIAS DE ESCAPE

Escape	Significado
\\	Caractere de contra barra
\'	Cota simples
\"	Cota dupla
\n	Nova linha
\r	Retorno de carro
\t	Tabulação

RAW STRINGS (CRUAS, SEM TRATAMENTO)

- Pode ser mais conveniente, quando trabalhando com escape, usar um formato de strings do tipo raw.
- Raw string é criado com o caractere (r) antes da string
  - Ex: Caso precisemos criar uma string para um caminho

```
>>> s = 'c:\novo\texto.txt'
>>> print(s)
c:
ovo    exto.txt
>>> w = 'c:\\novo\\texto.txt'
>>> z = r'c:\novo\texto.txt'
>>> print(z)
c:\novo\texto.txt
```

COTA TRIPLA

- Por conveniência, podemos criar string que ocupam mais de uma linha usando as cotas triplas

```
>>> s = ''' Sempre olhe
...     pelo lado
... bom da vida '''
>>> s
'sempre olhe\n    pelo lado\n bom da vida '
>>> print(s)
Sempre olhe
    pelo lado
bom da vida
```

OPERAÇÕES BÁSICAS COM STRINGS

- Tamanho de uma string

```
>>> s = 'casa'
>>> len(s)
4
```

- Concatenação e repetição

```
>>> s = 'agua' + 'salgada'
>>> print(s)
aguasalgada
>>> s = 'abc'
>>> print(4*s)
abcabcabcabc
```

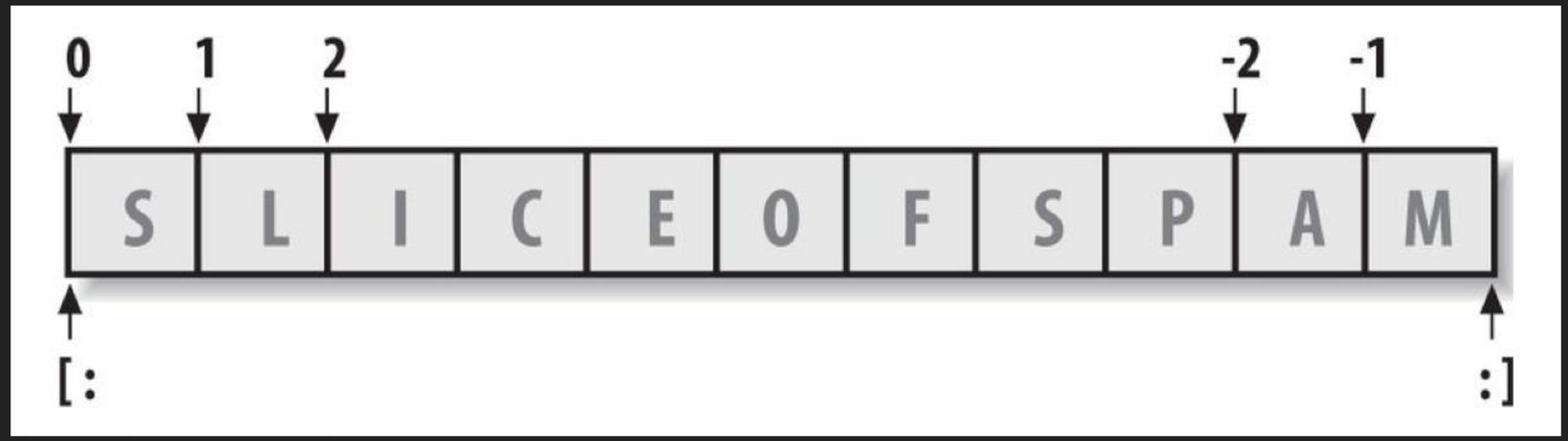
ITERAÇÕES EM UMA STRING

- É possível usar um laço for para iterar sobre os elementos da string

```
>>> s = 'python'
>>> for i in s:
...     print(i, end=' ')
p y t h o n
>>> 't' in s
True
>>> 'z' in s
False
```

INDEXAÇÃO E SLICE (RECORTE)

- Como uma string é uma sequencia de elementos, é possível acessá-los por um índice



- Ex:

```
>>> s = 'python'
>>> s[0], s[-1], s[1:4]
('p', 'n', 'yth')
```

INDEXAÇÃO E SLICE (RECORTE)

- Os elementos iniciam com índice ‘0'
- Elementos negativos representam a ordem da direita pra esquerda, iniciando com o último elemento sendo ‘-1’
- No slice, o limite inferior é inclusivo e o limite superior é não-inclusivo
- Existe o terceiro parâmetro no slice que representa o passo que, por padrão é 1. Por exemplo, X[i:j:k], i representa o limite inferior, j representa o limite superior e k representa o passo.

```
>>> s = ‘abcdefghijklmnop’
>>> s[1:10:2]
'bdfhj'
>>> s[::2]
'acegikmo'
>>> s = 'hello'
>>> s[::-1]
'olleh'
```

CONVERSÃO DE STRINGS

- Para utilizar uma string em uma operação matemática é preciso convertê-la antes

```
>>> s = '42'
>>> x = s + 5
Traceback (most recent call last):
  File "<pyshell#36>", line 1, in <module>
    x = s + 5
TypeError: Can't convert 'int' object to str implicitly
>>> x = int(s) + 5    # Soma aritmética
>>> x
47
>>> x = s + str(5)    # Concatenação
>>> x
425
```

STRINGS SÃO IMUTÁVEIS

- Não é possível alterar uma string

```
>>> s = 'python'
>>> s[1] = 'i'
Traceback (most recent call last):
  File "<pyshell#40>", line 1, in <module>
    s[1] = 'i'
TypeError: 'str' object does not support item assignment
>>> s = s[0] + 'i' + s[2:]
>>> s
pithon
>>> s = s.replace('y','i')
>>> s
pithon
```

MÉTODOS DE STRINGS

- Um objeto do tipo string tem vários métodos úteis para operações rotineiras

```
>>> s = 'python'

>>> dir(s)

['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
'__gt__', '__hash__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs',
'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace',
'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

MÉTODOS DE STRINGS

- Buscar e substituir dentro de uma string

```
>>> s = ‘python’
>>> s.find(‘th’)    # posição da primeira ocorrência
2
>>> s = 'xxxSPAMxxxEGGSxxx'
>>> s.replace(‘xxx’,’ooo’)  # substitui todas as ocorrências
'oooSPAMoooEGGSooo'
>>>s.replace(‘xxx’,’ooo’,1)  # substitui a primeira ocorrência
'oooSPAMxxxEGGSxxx'
```

CONVERSÃO ENTRE STRING E LISTA

- Em diversos casos é útil tratar listas como strings, ou o inverso

```
>>> s = ‘python'
>>> L = list(s)
>>> L
[‘p’,’y’,’t’,'h','o','n']
>>> L[0]
'p'
>>> w = ‘’.join(L)
>>> w
'python'
```

CONVERSÃO ENTRE STRING E LISTA

- Com o uso do método split é possível usar padrões de separação de strings para transformá-la em uma lista

```
>>> s = ‘joao,paulo,pedro,marco’

>>> L = s.split(‘,')

>>> L

[‘joao’,’paulo’,’pedro’,'marco']
```

FORMATAÇÃO DE STRINGS

- O Python permite através de uma expressão com sintaxe específica, fazer substituições de strings com padrões de formatação.
  - Ex:

```
>>> nome = 'joao'

>>> ‘a casa de %s fica em natal’ % nome

‘a casa de joao fica em natal’
```

FORMATAÇÃO DE STRINGS

- A sintaxe completa da expressão é:
  - %[(nomedachave)] [flags] [tamanho] [.precisão] código
  - Aonde:
    - (nome da chave) - Caso se utilize dicionários para os valores
    - flags - Especifica alguns aspectos, tais como, justificação (-), sinais numéricos (+), números positivos sem sinal e negativos com sinal (espaço), preenchimento com zeros
    - tamanho - Tamanho minimo do texto substituído
    - precisão - Para ponto flutuante, o número de casas decimais
    - Código - Código que representa o tipo de dado a ser substituído

FORMATAÇÃO DE STRINGS - CÓDIGOS MAIS COMUNS

Código	Significado
s	String
d	decimal
i	inteiro
x	hexadecimal
e	Ponto flutuante com expoente
f	Ponto flutuante

FORMATAÇÃO DE STRINGS

- Exemplos

```
>>> x = 1234
>>> 'inteiros: ... %d ... %-6d ... %06d' % (x, x, x)
'inteiros: ... 1234 ... 1234 ... 001234'
>>> x = 1.23456789
>>> '%-6.2f | %05.2f | %+06.1f' % (x, x, x)
'1.23 | 01.23 | +001.2'
```

LISTAS

- Listas são coleções ordenadas de objetos
- Listas podem conter quaisquer tipos de objetos
- Listas são Objetos Mutáveis (Podem ter seus elementos alterados)
- Listas têm tamanho variável, são heterogêneas e arbitrariamente aninhadas
  - Pode-se adicionar ou remover elementos
  - Podem conter tipos diferentes
  - Podem conter outras listas como elementos (Equivalente aos arrays)

COMO CRIAR LISTAS

- Existem formas variadas para criação de uma lista
  - Lista vazia

```
>>> L = []
```

```
>>> L = list()
```

  - Lista com itens

```
>>> L = [4, ‘abc’, 1.23, [9, 8, 7]]
```

  - Lista a partir de um objeto iterável

```
>>> L = list(‘spam')
```

```
>>> L = list(range(0, 10))
```

INDEXAÇÃO E SLIDE (RECORTE)

- Como listas são sequências, as operações de indexação e slide são iguais as mesmas operações com strings

```
>>> L = [4, 'abc', 1.23]
>>> L[0]
4
>>> L[-2]
'abc'
>>> L[:2]
[4, 'abc']
```

OPERAÇÕES BÁSICAS COM LISTAS

- As operações básicas também têm efeito parecido com strings

```
>>> L = [4, 'abc', 1.23]
>>> M = ['fgh', 5.6, 10, 145]
>>> len(L)
3
>>> L + M
[4, 'abc', 1.23, 'fgh', 5.6, 10, 145]
>>> 3*L
[4, 'abc', 1.23, 4, 'abc', 1.23, 4, 'abc', 1.23]
```

ITERAÇÃO E ASSOCIAÇÃO

- Associação é utilizada para testar se um objeto pertence a lista

```
>>> L = [4, ‘abc’, 1.23]
>>> ‘abc’ in L
True
>>> for x in L:
...     print(x)
...
4
abc
1.23
```

COMPREENSÃO DE LISTAS (LIST COMPREHENSIONS)

- List Comprehensions é um recurso para criação de listas por iteração a outro objeto

```
>>> res = [c*4 for c in 'abc']

>>> res

['aaaa', 'bbbb', 'cccc']
```

MATRIZES

- Listas podem ser utilizadas para representar matrizes

```
>>> M = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> M[1]
[4, 5, 6]
>>> M[2][1]
8
>>> V = [x[1] for x in M]
[2, 5, 8]
```

1	2	3
4	5	6
7	8	9

ALTERAÇÃO DE LISTAS

- Através de indexação e slides

```
>>> L = [1, 5, 6]
>>> L[1] = 9
>>> L
[1, 9, 6]
>>> L[0:2] = ['abc', 25]
>>> L
['abc', 25, 6]
```

ALTERAÇÃO DE LISTAS

- Através de indexação e slides

```
>>> L = [1, 5, 6]
>>> L[1:1] = [9, 10]
>>> L
[1, 9, 10, 5, 6]
>>> L[1:2] = []
>>> L
[1, 10, 5, 6]
```

ALTERAÇÃO DE LISTAS

- Através de métodos

```
>>> L = [1, 5, 6]
>>> L.append('abc')      # Adiciona um elemento ao final
>>> L
[1, 5, 6, 'abc']
>>> L.extend([10, 11])   # Adiciona vários itens ao final
>>> L
[1, 5, 6, 'abc', 10, 11]
```

ALTERAÇÃO DE LISTAS

- Através de métodos

```
>>> L = [1, 5, 6, 7, 9]
>>> L.pop()      # remove ultimo elemento
9
>>> L
[1, 5, 6, 7]
>>> L.pop(2)    # remove por posição
6
>>> L
[1, 5, 7]
```

ALTERAÇÃO DE LISTAS

- Através de métodos

```
>>> L = [1, 9, 6, 5, 2]
>>> L.reverse()      # inverte ordem
>>> L
[2, 5, 6, 9, 1]
>>> L.remove(6)    # remove por conteúdo
>>> L
[2, 5, 9, 1]
>>> L.sort()    # ordena a lista
>>> L
[1, 2, 5, 9]
```

DICIONÁRIOS

- Dicionários são coleções não ordenadas de objetos
  - Itens em um dicionário são armazenados e acessados através de uma chave
- Dicionário são Objetos Mutáveis (Podem ter seus elementos alterados)
- Dicionários têm tamanho variável, são heterogêneas e arbitrariamente aninhados
  - Pode-se adicionar ou remover elementos
  - Podem conter tipos diferentes
  - Podem conter listas ou outros dicionários como elementos
- São equivalentes aos Registros em outras linguagens

COMO CRIAR DICIONÁRIOS

- Existem formas variadas para criação de um dicionário
  - Dicionário vazio

```
>>> D = {}
```

```
>>> D = dict()
```

  - Dicionário com itens

```
>>> D = {'nome': 'joao', 'idade': 25}
```

```
>>> D = dict(nome='joao', idade=25)
```

  - Dicionário a partir pares de chave/valor

```
>>> D = dict([('nome','idade'),('joao',25)])
```

INDEXAÇÃO DE DICIONÁRIOS

- Elementos de um dicionário devem ser acessados através de sua chave

```
>>> D = {'nome': 'joao', 'idade': 25}
>>> D['nome']
'joao'
>>> D['idade']
25
>>> len(D)
2
```

ASSOCIAÇÃO EM DICIONÁRIOS

```
>>> D = {‘nome’: ‘joao’, ‘idade’: 25}

>>> ‘joao' in D

True
```

ALTERAÇÃO DE DICIONÁRIOS

```
>>> D = {'nome': 'joao', 'idade': 25}
>>> D['sexo'] = 'masculino'     # Adiciona elemento
>>> D
{'nome': 'joao', 'idade': 25, 'sexo': 'masculino'}
>>> D['idade'] = 30     # Edita elemento
>>> D
{'nome': 'joao', 'idade': 30, 'sexo': 'masculino'}
>>> del D['idade']     # Apaga elemento
>>> D
{'nome': 'joao', 'sexo': 'masculino'}
```

ALTERAÇÃO DE DICIONÁRIOS

```
>>> D = {'nome': 'joao', 'idade': 25}
>>> D.pop()
Traceback (most recent call last):
  File "<pyshell#62>", line 1, in <module>
    D.pop()
TypeError: pop expected at least 1 arguments, got 0
>>> D.pop('idade')
25
>>> D
{'nome': 'joao'}
```

MÉTODOS DE DICIONÁRIOS

```
>>> D = {'nome': 'joao', 'idade': 25}
>>> list(D.keys())    # Cria lista com as chaves
['nome', 'idade']
>>> list(D.values())  # Cria lista com os valores
['joao', 25]
>>> list(D.itens())    # Cria lista com os pares chave/valor
[('nome', 'joao'), ('idade', 25)]
```

MÉTODOS DE DICIONÁRIOS

```
>>> D = {‘nome’: ‘joao’, ‘idade’: 25}
>>> D[‘sexo’]
Traceback (most recent call last):
  File "<pyshell#56>", line 1, in <module>
    D['sexo']
KeyError: 'sexo'
>>> D.get(‘sexo')   # Retorna None caso a chave não exista
>>>
```

ITERAÇÃO EM DICIONÁRIOS

```
>>> D = {'nome': 'joao', 'idade': 25}
>>> for i in D.keys():
...     print(i)
...
nome
idade
```

TUPLAS

- Tuplas são coleções ordenadas de objetos
- Tuplas têm algumas similaridades com listas, porém, com menos métodos disponíveis.
- Tuplas são Objetos Imutáveis (Não podem ter seus elementos alterados)
- Tuplas têm tamanho fixo, são heterogêneas e arbitrariamente aninhadas
  - Podem conter tipos diferentes
  - Podem conter outras listas como elementos (Equivalente aos arrays)

COMO CRIAR TUPLAS

- Existem formas variadas para criação de uma Tupla
  - Tupla vazia

```
>>> T = ()
```

```
>>> T = tuple()
```

  - Tupla com itens

```
>>> T = (4,)
```

```
>>> T = (4, 'abc', 1.23, (9, 8, 7))
```

```
>>> T = 10, 20
```

  - Tupla a partir de um objeto iterável

```
>>> T = tuple('spam')
```

```
>>> T = tuple(range(0, 10))
```

POR QUE TUPLAS?

- A semelhança com listas é evidente
- Por que existe Tuplas se Listas têm mais funcionalidades?
  - Tuplas é um termo originado da Matemática e portanto foi pensada com esse objetivo
  - Tuplas são imutáveis e, portanto, mantém integridade dos objetos
  - Podem ser usadas aonde listas não podem:
    - Chaves de Dicionários
    - Algumas built-in exigem o uso de Tuplas

INDEXAÇÃO E SLIDE (RECORTE)

- Como Tuplas são sequências, as operações de indexação e slide são iguais as mesmas operações com listas

```
>>> T = (10, 3.56, 200, ’casa’)
>>> T[0]
10
>>> T[-1]
‘casa’
>>> T[1:3]
(3.56, 200)
>>> T[2:]
(200, 'casa')
```

OPERAÇÕES BÁSICAS COM TUPLAS

- Concatenação

```
>>> (1, 2) + (3, 4)
(1, 2, 3, 4)
```

- Repetição

```
>>> (1, 2) * 3
(1, 2, 1, 2, 1, 2)
```

ITERAÇÃO E ASSOCIAÇÃO

- Associação é utilizada para testar se um objeto pertence a tupla

```
>>> T = (10, 20, 30)
>>> 10 in T
True
>>> for x in T:
...     print(x)
...
10
20
30
```

COMPREENSÃO DE TUPLAS (TUPLE COMPREHENSIONS)

- `Tuple Comprehensions é um recurso para criação de tuplas por iteração a outro objeto

```
>>> s = 'ifrn'

>>> res = tuple(c*2 for c in s)

>>> res

('ii', 'ff', 'rr', 'nn')
```

PECULARIEDADE

- O uso de parênteses pode confundir quando trabalhando com tuplas

```
>>> x = (10)          # O inteiro 10
>>> x
10
>>> y =(10,)          # A tupla com um elemento inteiro
>>> y
(10,)
```

CONVERSÃO

- Em alguns casos é necessário a conversão entre tipos. Por exemplo, como tuplas são imutáveis, para ordená-la é necessário convertê-la para lista

```
>>> T = (‘c’, ‘a’, ‘x’, ‘r’)
>>> tmp = list(T)
>>> tmp.sort()
>>> tmp
[‘a’, ‘c’, ‘r’, ‘x’]
>>> T = tuple(tmp)
>>> T
(‘a’, ‘c’, ‘r’, ‘x’)
```

MÉTODOS

- Tuplas têm um número limitado de métodos

```
>>> T = (1, 2, 3, 2, 5, 2)
>>> T.index(2)       # offset do primeiro elemento com valor 2
1
>>> T.index(2, 2) # próximo indice com valor 2 após o índice 2
3
>>> T.count(2)  # Quantas vezes o valor 2 aparece
3
```

FILES

- O Tipo built-in File possibilita uma forma de acessar arquivos armazenados no computador, a partir do código Python
- São criados através da função built-in “*open"* , que cria um objeto do tipo File para referenciar um arquivos no sistema de arquivos do computador
- Usando Arquivos
  - A utilização de Iterators são a melhor forma de ler e escrever arquivos
  - O conteúdo lido dos arquivos são sempre retornados na forma de string
  - As operações de escritas de arquivos são buferizadas e o conteúdo dos arquivos são "buscáveis"
  - A operação de fechar o arquivo é opcional. O Python realiza o Auto-close.

ABRINDO ARQUIVOS

- Para abrir um arquivo

obj_arquivo = open(filename, mode='rt', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)

  - filename: Nome do arquivo com caminho (relativo ou absoluto)
  - mode: Modo em que o arquivo será aberto (Somente Leitura, Escrita, Adicionar, binário, etc) - Opcional
  - buffering: permite alterar políticas de buferização dos dados (ativar, desativar, tamanho, etc) - Opcional
  - encoding: Para o modo texto somente. define qual encoding será utilizado - Opcional
  - errors: Especifica como os erros de encoding serão tratados. Somente modo texto - Opcional
  - newline: Controla como newlines serão inseridos (None, '', '\r', '\n', '\r\n') - Opcional
  - closefd: Mantém o descritor de arquivo aberto quando o arquivo é fechado - Opcional
  - opener: Repassa para uma função customizada o descritor do arquivo - Opcional

MODO DE ABERTURA DE ARQUIVOS

- Valores possíveis do parâmetro mode:

caractere	significado
r	*Abre somente leitura*
w	*Abre para escrita; truncamento do arquivo*
x	*Cria um novo arquivo e abre-o pra escrita*
a	*Abre para escrita; apenando no final*
b	*Modo binário*
t	*Modo Texto*
+	*Abre um arquivo para atualização*

UTILIZANDO ARQUIVOS

- Abrindo um arquivo para escrita, escrevendo duas linhas e fechando o arquivo

```
>>> arq = open('meuarquivo.txt', 'w')
>>> arq.write('Ola Mundo, arquivo Texto.\n')
26
>>> arq.write('Tchau Mundo!!!\n')
15
>>> arq.close()
```

- Abrindo um arquivo para leitura e lendo o seu conteúdo

```
>>> arq = open('meuarquivo.txt')
>>> arq.readline()
'Ola Mundo, arquivo Texto.'
>>> arq.readline()
'Tchau Mundo!!!'
>>> arq.readline()
''
```

LENDO ARQUIVOS COM ITERATORS

- A forma mais prática para ler as linhas d num arquivo é com um iterator

```
>>> arq = open(‘meuarquivo.txt’, 'w')
>>> for linha in arq:
...    print(linha, end=‘')
...
‘Ola Mundo, arquivo Texto.’
‘Tchau Mundo!!!’
```

ARQUIVO BINÁRIO VERSUS ARQUIVO TEXTO

- Modo Texto
  - Os conteúdos são tratados como strings, é realizado o encoding e decoding automaticamente e traduzem o caractere de fim-de-linha
- Modo Binário
  - Os conteúdos representam “bytes strings” e permitem o acesso de forma inalterada

ARMAZENANDO OBJETOS EM ARQUIVOS

- Usando estratégias de conversão
  - Conversão para string antes da escrita
  - Avaliação (evaluation) após leitura
- Usando armazenamento nativo - pickle
  - Modulo específico para armazenamento de objetos em arquivos
  - Conversão automática
- Usando formato JSON
  - Formato de serialização independente de linguagem
  - Suportado por diversos sistemas e linguagens

ARMAZENAMENTO USANDO CONVERSÃO

- Para armazenar objetos em um arquivo texto
- Os objetos devem ser convertidos em strings
- a operação inversa, de recuperar os objetos do arquivo texto, implica em convertê-los de volta de strings para objetos
- Quando não for possível usar o construtor do tipo para converter, utiliza-se o método “eval" para avaliar a string e gerar o objeto (Como se fosse a avaliação de uma digitação daquela sequência de caracteres no Interpretador Python)

ARMAZENAMENTO USANDO CONVERSÃO

- Exemplo de script para armazenamento

```
>>> x, y, z = 43, 44, 45
>>> s = 'spam'
>>> d = {'a':1, 'b':2}
>>> l = [1, 2, 3]
>>> f = open('datafile.txt', 'w')
>>> f.write(s + '\n')
>>> f.write('%s, %s, %s\n' % (x, y, z))
>>> f.write(str(l) + '$' + str(d) + '\n')
>>> f.close()
```

ARMAZENAMENTO USANDO CONVERSÃO

- Exemplo de script para leitura

```
>>> chars = open('datafile.txt').read()
>>> chars
"spam\n43, 44, 45\n[1, 2, 3]${'a': 1, 'b': 2}\n"
>>> print(chars)
spam
43, 44, 45
[1, 2, 3]${'a': 1, 'b': 2}\n"
```

ARMAZENAMENTO USANDO CONVERSÃO

- Convertendo de volta os objetos

```
>>> arquivo = open('datafile.txt')
>>> linha = arquivo.readline()
>>> linha.rstrip()
"spam"
>>> linha = arquivo.readline()
>>> lista_numeros = [int(p) for p in linha.split(',')]
>>> lista_numeros
[43, 44, 45]
>>> linha = arquivo.readline()
>>> sequencias = [eval(p) for p in linha.split('$')]
>>> sequencias
[[1, 2, 3],{'a': 1, 'b': 2}]
```

ARMAZENAMENTO USANDO O MÓDULO PICKLE

- O uso do “eval" é poderoso, contudo, se nas strings lidas do arquivo contiver algum código interpretável, isso pode representar sérios problemas de segurança
- O módulo pickle permite armazenar qualquer objeto python em um arquivo e recuperá-lo, sem riscos de segurança, e sem necessidade de conversões explícitas de e para strings
- Como pickle usa uma codificação proprietária, que não é string, o tipo de arquivo adequado para armazenar o conteúdo gerado pelo pickle é o binário

ARMAZENAMENTO USANDO O MÓDULO PICKLE

- Exemplo de script para armazenamento

```
>>> import pickle
>>> d = {'a':1, 'b':2}
>>> f = open('datafile.pkl', 'wb')
>>> pickle.dump(d, f)
>>> f.close()
```

- Exemplo de script para armazenamento

```
>>> open('datafile.pkl').read()
'€\x03}q\x00(X\x01\x00\x00\x00aq\x01K\x01X\x01\x00\x00\x00bq\x02K\x02u.'
```

ARMAZENAMENTO USANDO O MÓDULO PICKLE

- Exemplo de script para recuperar os objetos

```
>>> import pickle
>>> f = open(‘datafile.pkl’, 'rb')
>>> e = pickle.load(f)
>>> e
{'a':1, 'b':2}
```

ARMAZENAMENTO USANDO O FORMATO JSON

- Formato emergente para troca de dados
- Suportado por várias linguagens e sistemas
  - Ex: MongoDB utiliza JSON para armazenar os dados em sua base de dados
- Apesar de não suportar todos os recursos que o módulo pickle implementa, a portabilidade do JSON pode ser uma grande vantagem
- O formato de serialização do JSON já bem próximo do formato de representação dos objetos python

ARMAZENAMENTO USANDO O FORMATO JSON

- Exemplo de script para armazenamento

```
>>> import json
>>> name = dict(first='bob', last='smith')
>>> registro = dict(name=name, job=['dev', 'mgr'], idade=45)
>>> registro
{'job': ['dev', 'mgr'], 'name': {'first': 'bob', 'last': 'smith'}, 'idade': 45}
>>> f = open('datafile.json', 'w')
>>> json.dump(registro, f, indent=4)
>>> f.close()
```

ARMAZENAMENTO USANDO O FORMATO JSON

- Exemplo de script para recuperar os objetos

```
>>> import json
>>> f = open('datafile.json')
>>> e = json.load(f)
>>> e
{'job': ['dev', 'mgr'], 'name': {'first': 'bob', 'last': 'smith'}, 'idade': 45}
```