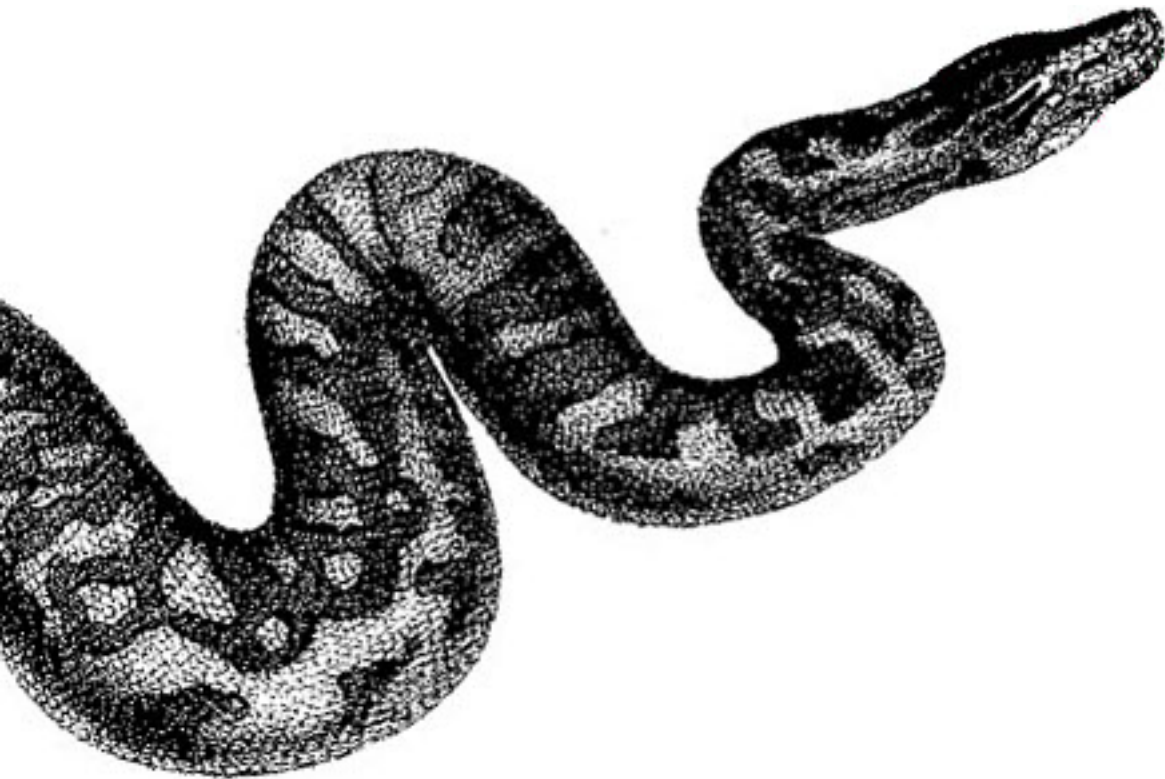


Python: apresentando a sintaxe



Luciano Ramalho
luciano@ramalho.org

Blocos por endentação

dois-pontos marca o início do bloco

indentação dentro do bloco deve ser constante*

```
for letra in 'Python':  
    cod = ord(letra)  
    print cod, letra  
print 'FIM'
```

retorno ao nível anterior de indentação marca o final do bloco

* por convenção, usa-se 4 espaços por nível (mas basta ser consistente; evite usar tabs)

Em vez de *Hello World...*

```
# coding: utf-8
from time import sleep, strftime

while True:      # rodar para sempre
    hora = strftime( '%H:%M:%S' )
    print hora
    sleep(1)     # aguardar 1 segundo
```

Primeiro programa

- salvar com o nome de hora.py
- executar:

```
$ python hora.py
```

-
- transformar em executável (Linux ou Mac OS X)

- incluir esta linha no topo do arquivo:

```
#!/usr/bin/env python
```

- executar este comando no shell:

```
$ chmod +x hora.py
```

Comentários

- O símbolo **#** indica que o texto partir daquele ponto e até o final da linha deve ser ignorado pelo interpretador **python**
 - exceto quando **#** aparece em uma string
- Para comentários de várias linhas, usa-se três aspas simples ou duplas (isso cria uma “doc string” e não é ignorada pelo **python**, mas é usada para documentar)

```
""" Minha terra tem palmeiras,  
Onde canta o Sabiá;  
As aves, que aqui gorjeiam,  
Não gorjeiam como lá. """
```

Comentários especiais

- Para tornar o script executável em sistemas UNIX:

```
#!/usr/bin/env python
```

- `#!` precisam ser os primeiros dois bytes do arquivo (`#!` é pronunciado *sheh-bang*)
- Para indicar uso de uma codificação não-ASCII no código-fonte:

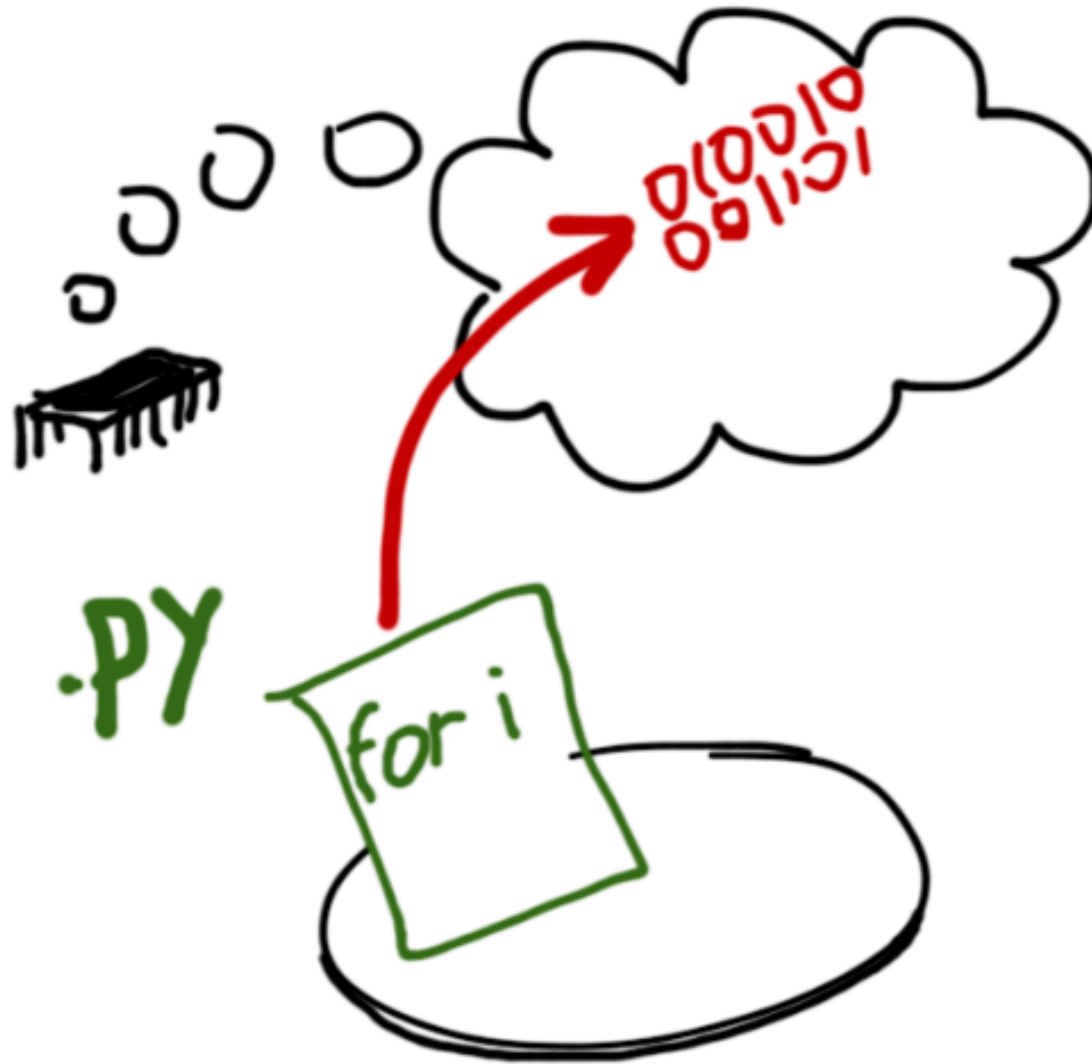
```
#coding: utf-8
```

- tem que ser a primeira linha do arquivo, mas pode ser a segunda após `#!`

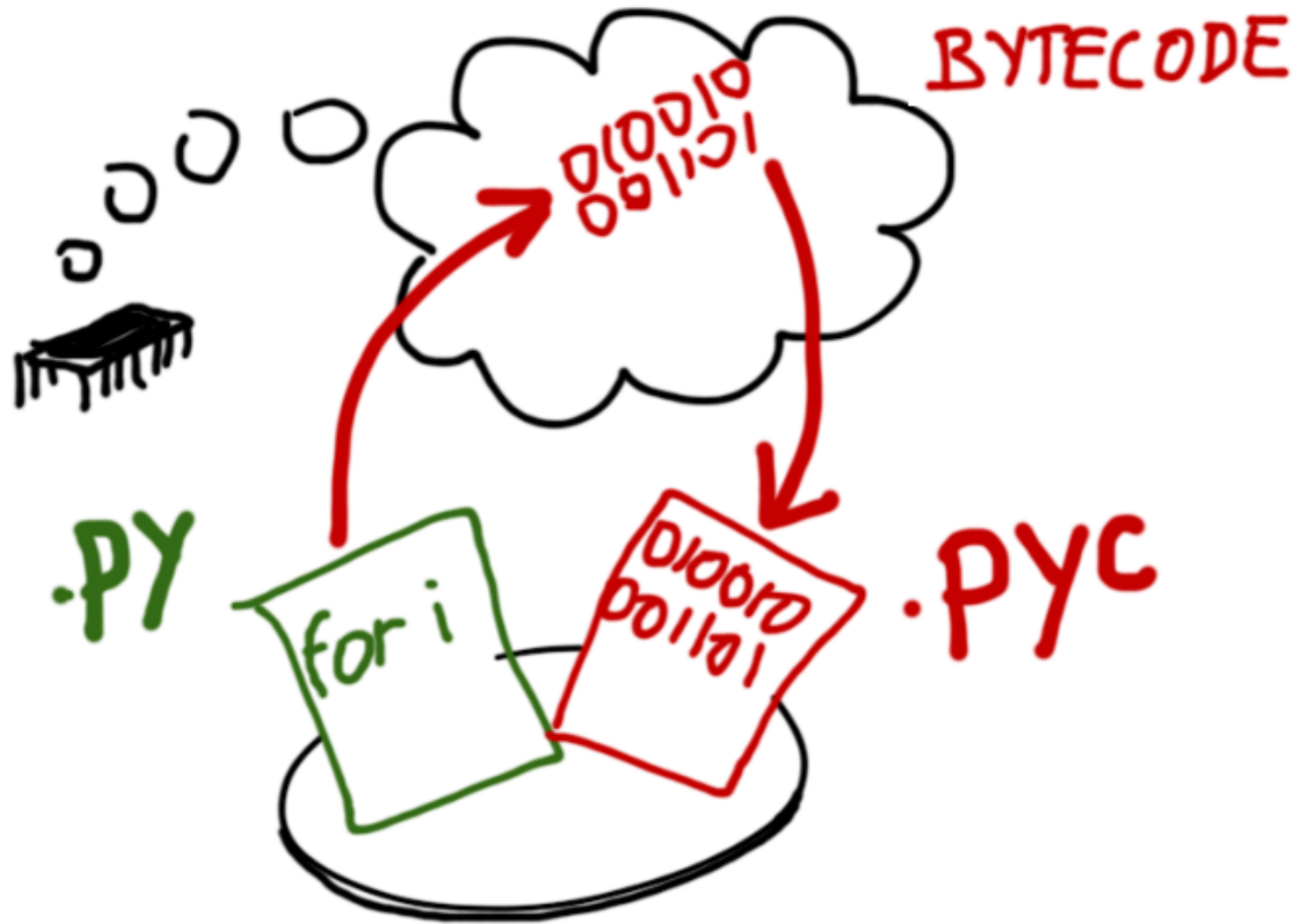
Como Python executa um script



Como Python executa um script



Como Python executa um script



Blocos no console interativo

```
>>> for letra in 'Python':  
...     cod = ord(letra)  
...     print cod, letra  
...  
80 P  
121 y  
116 t  
104 h  
111 o  
110 n  
>>>
```

* por convenção, usa-se 4 espaços por nível
(mas no console, muita gente usa 2 espaços)

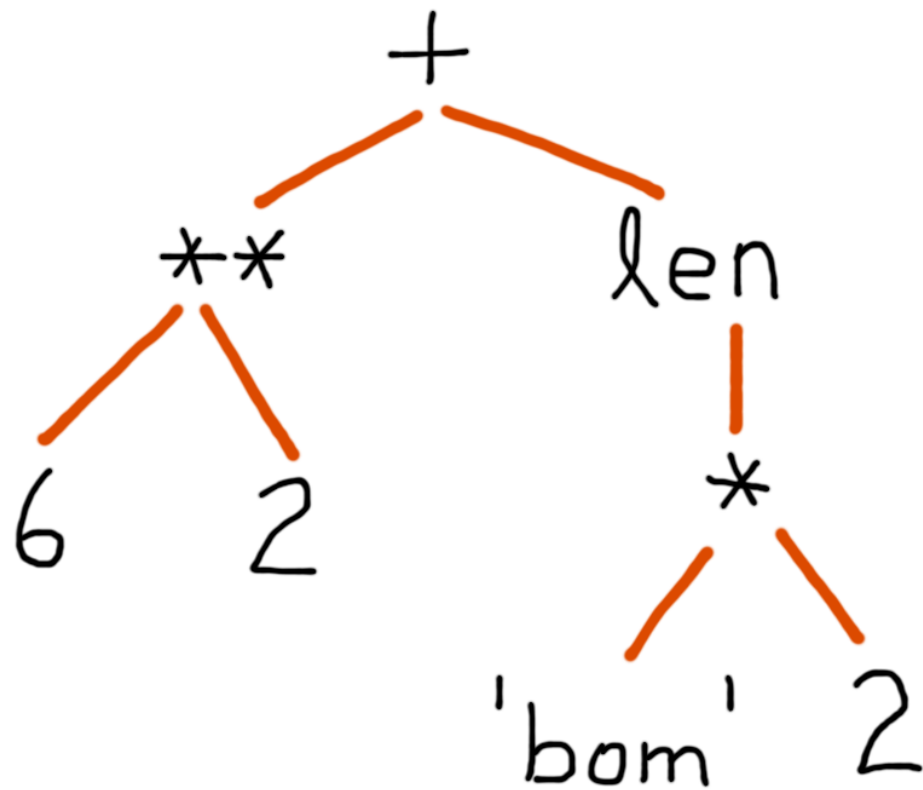
O que acontece no console interativo

```
>>> 6**2 + len('bom'*2)
```

O que acontece no console interativo

```
>>> 6**2 + len('bom'*2)
```

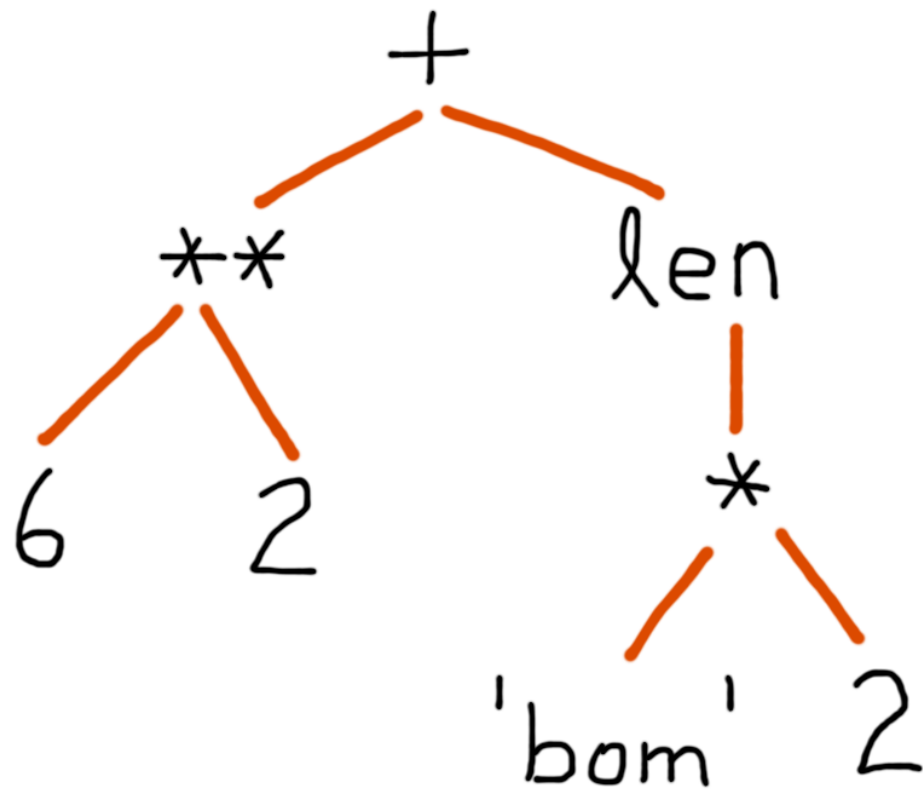
Read



O que acontece no console interativo

```
>>> 6**2 + len('bom'*2)
```

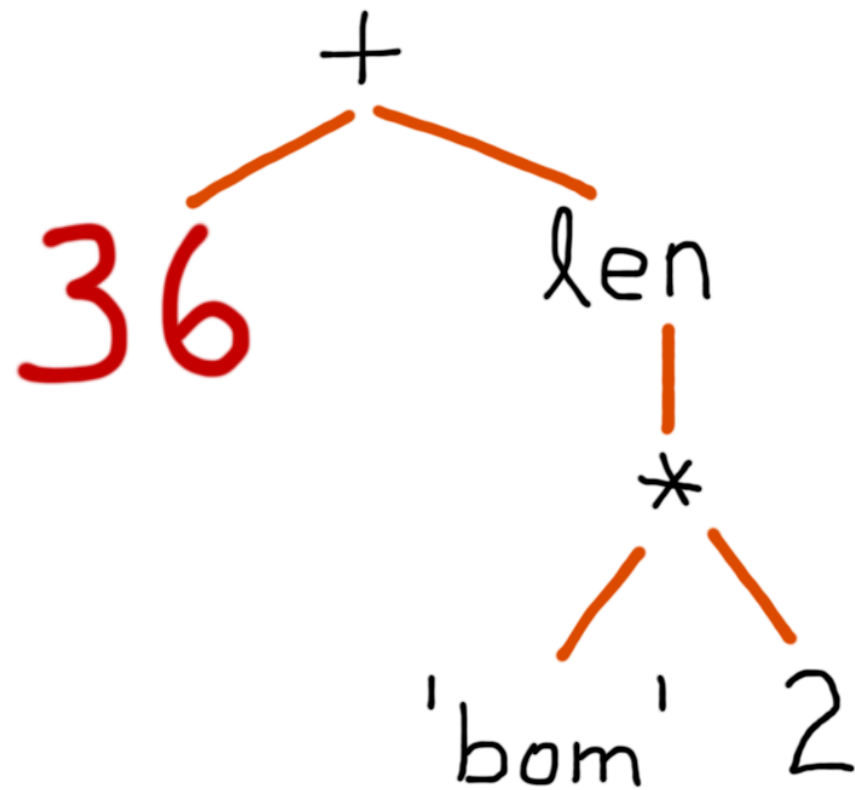
Eval



O que acontece no console interativo

```
>>> 6**2 + len('bom'*2)
```

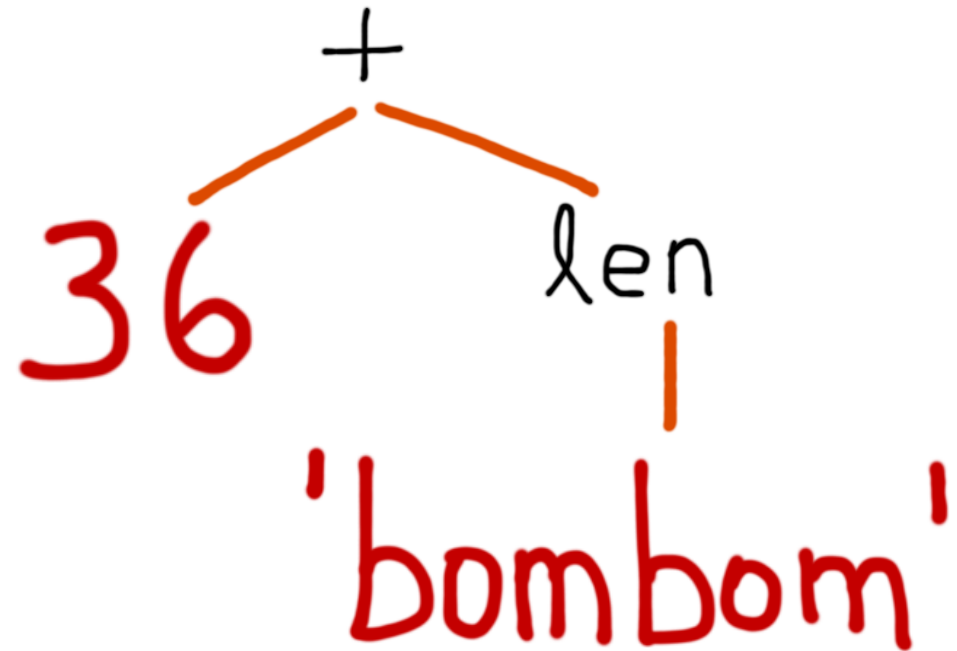
Eval



O que acontece no console interativo

```
>>> 6**2 + len('bom'*2)
```

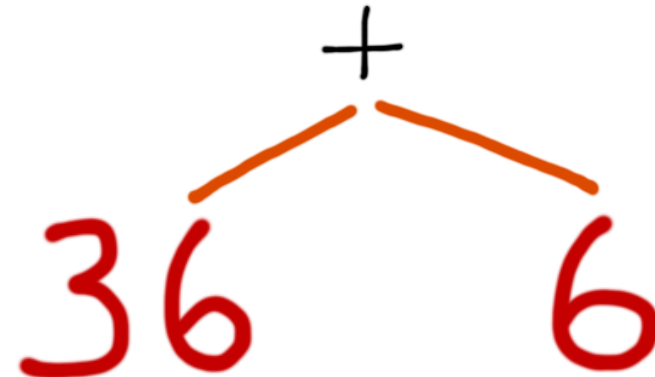
Eval



O que acontece no console interativo

```
>>> 6**2 + len('bom'*2)
```

Eval



O que acontece no console interativo

```
>>> 6**2 + len('bom'*2)
```

42

Eval

O que acontece no console interativo

```
>>> 6**2 + len('bom'*2)  
42
```

Print

42

O que acontece no console interativo

```
>>> 6**2 + len('bom'*2)
```

```
42
```

```
>>>
```

Loop



O que acontece no console interativo

```
>>> 6**2 + len('bom'*2)  
42
```

Read
Eval
Print
Loop



Blocos

- Todos os comandos que aceitam blocos:
 - **if/elif/else**
 - **for/else**
 - **while/else**
 - **def**
 - **try/except/finally/else**
 - **class**
 - **with**
- Se o bloco tem apenas um comando, pode-se escrever tudo em uma linha:

```
if n < 0: print 'Valor inválido'
```

Tipos de dados básicos

- Números: int, long, float, complex
- Strings: str e unicode
- Listas e tuplas: list, tuple
- Dicionários: dict
- Arquivos: file
- Booleanos: bool (True, False)
- Conjuntos: set, frozenset
- None

Números inteiros

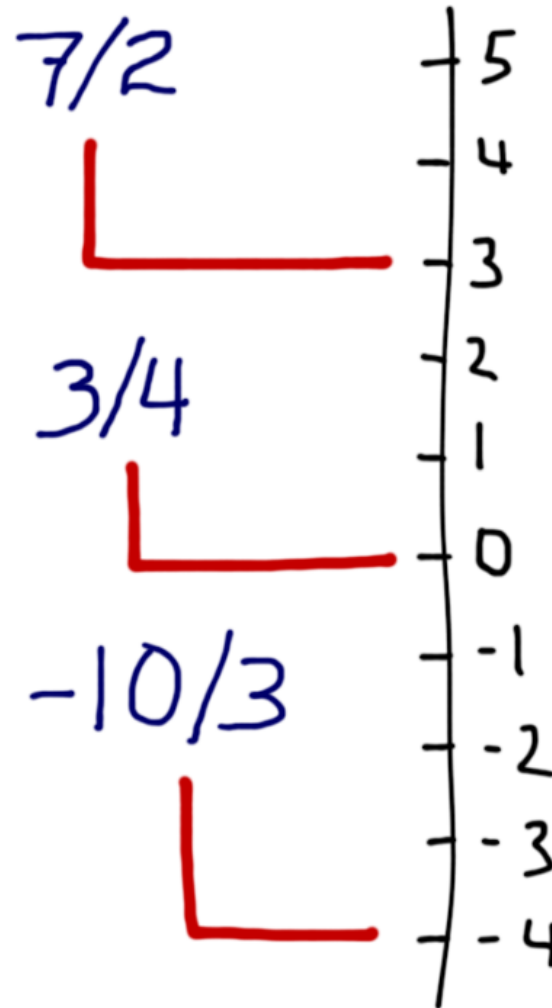
- **int**: usualmente inteiros de 32 bits
- **long**: alcance limitado apenas pela memória
- **ATENÇÃO**: a divisão entre inteiros em Python < 3.0 sempre retorna outro inteiro
- Python promove de **int** para **long** automaticamente

```
>>> 1 / 2
0
>>> 1. / 2
0.5
```

```
>>> 2**30 + (2**30 - 1)
2147483647
>>> 2**31
2147483648L
```

Divisão entre inteiros

```
>>> 7/2
3
>>> 3/4
0
>>> -10/3
-4
```



floor
division
||
divisão
pelo
piso

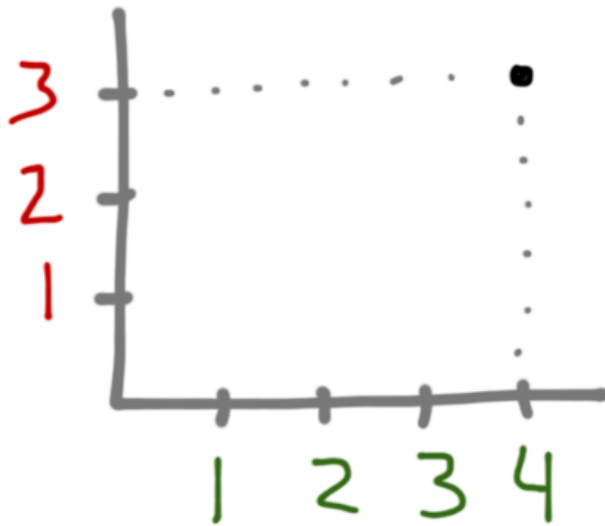
Ponto flutuante

- **float**: ponto-flutuante de 32 bits
- Construtores ou funções de conversão:
 - **int(x)**
 - **long(x)**
 - **float(x)**
 - **x** pode ser número ou string

Números complexos

- Construtor:
 - `complex(r, i)`

$$z = 4 + 3j$$



```
>>> z = 4 + 3j
>>> z.real
4.0
>>> z.imag
3.0
>>> z + (5 + 1.5j)
(9+4.5j)
>>> z * 2
(8+6j)
```

`z.real` → 4.0

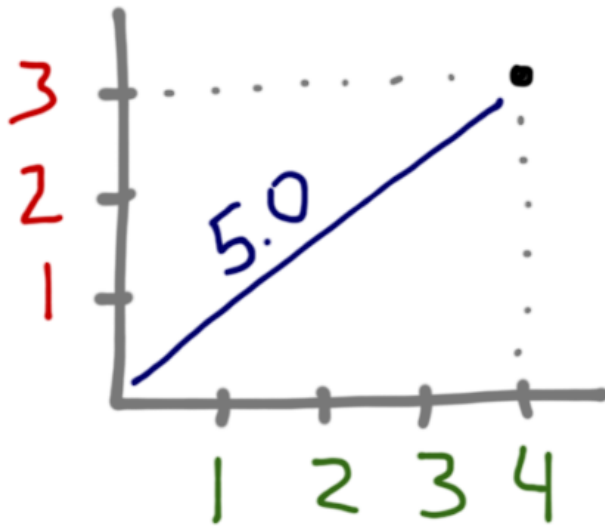
`z.imag` → 3.0

Números complexos

- Construtor:
 - `complex(r, i)`

$$z = 4 + 3j$$

```
>>> z = 4 + 3j
>>> abs(z)
5.0
```

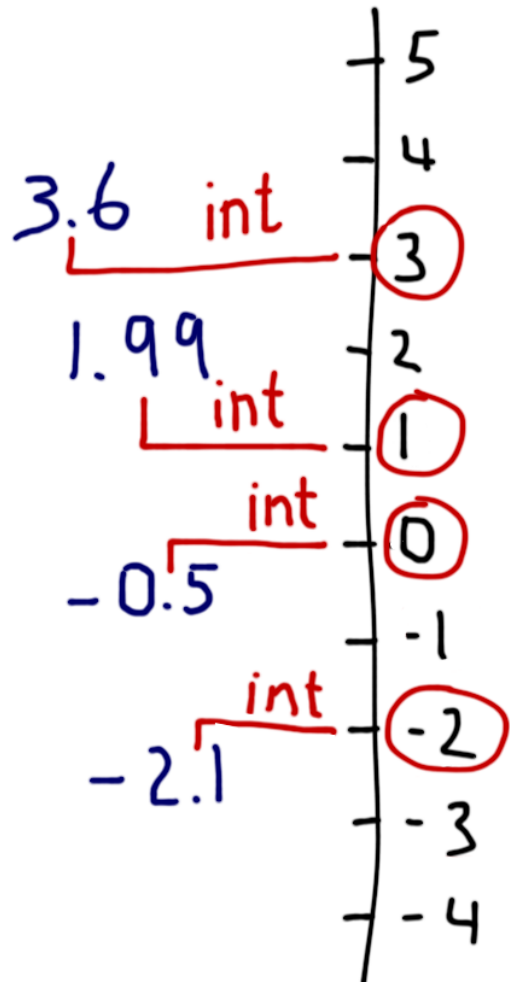


$$\text{abs}(z) \rightarrow 5.0$$

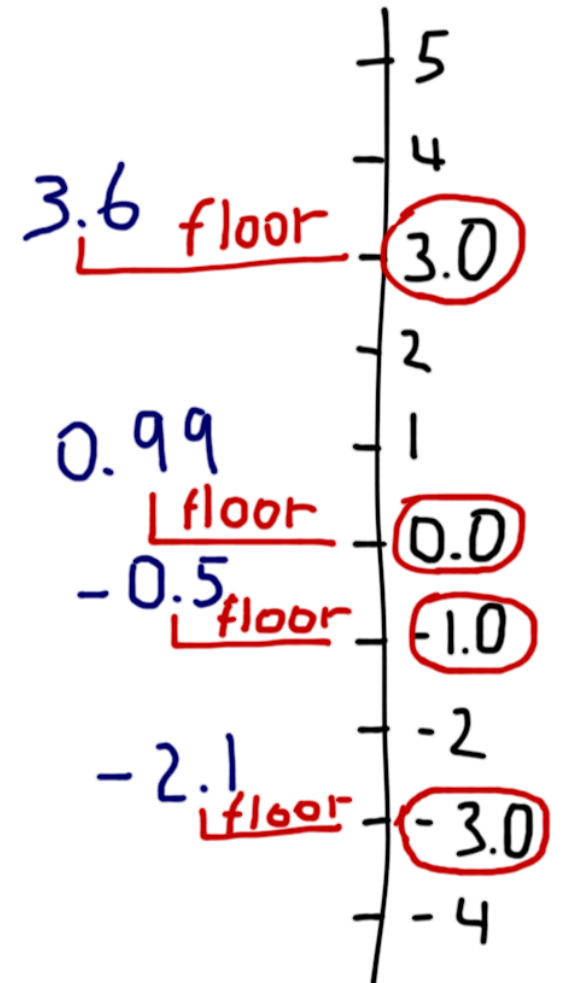
Operadores numéricos

- Aritméticos
 - básicos: `+` `-` `*` `/` `**` (o último: potenciação)
 - aritmética de inteiros: `%` `//` (resto e divisão)
- Bit a bit:
 - `&` `|` `^` `~` `>>` `<<` (and, or, xor, not, shr, shl)
- Funções numéricas podem ser encontradas em diversos módulos
 - principalmente o módulo `math`
 - `sqrt()`, `log()`, `sin()`, `pi`, `radians()`, `floor()` etc...

Construtor int x função floor



```
>>> int(3.6)
3
>>> int(1.99)
1
>>> int(0.5)
0
>>> int(-0.5)
0
>>> int(-2.1)
-2
>>> from math import floor
>>> floor(3.6)
3.0
>>> floor(0.99)
0.0
>>> floor(-0.5)
-1.0
>>> floor(-2.1)
-3.0
```



Booleans

- Valores: **True**, **False**
 - outros valores: conversão automática
- Conversão explícita: **bool(x)**

```
>>> bool(0)
False
```

```
>>> bool('')
False
```

```
>>> bool([])
False
```

```
>>> bool(3)
True
```

```
>>> bool('0')
True
```

```
>>> bool([[[]]])
True
```

Operadores booleanos

- Operadores relacionais

- `==` `!=` `>` `>=` `<` `<=` `is` `is not`

- Sempre retornam um **bool**

- Operadores lógicos

- **and** **or**

- Retornam o primeiro ou o segundo valor

- Exemplo: `print nome or '(sem nome)'`

- Avaliação com curto-circuito

- **not**

- sempre retorna um **bool**

None

- O valor nulo e único (só existe uma instância de **None**)
- Equivale a **False** em um contexto booleano
- Usos comuns:
 - valor default em parâmetros de funções
 - valor de retorno de funções que não têm o que retornar
- Para testar, utilize o operador **is**:

```
if x is None: return y
```


Strings

- **str**: cada caractere é um byte; acentuação depende do encoding
- strings podem ser delimitadas por:
 - aspas simples ou duplas: 'x', "x"
 - três aspas simples ou duplas: '''x''', """x"""

```
>>> fruta = 'maçã'
>>> fruta
'ma\xc3\xa7\xc3\xa3'
>>> print fruta
maçã
>>> print repr(fruta)
'ma\xc3\xa7\xc3\xa3'
>>> print str(fruta)
maçã
>>> len(fruta)
6
```

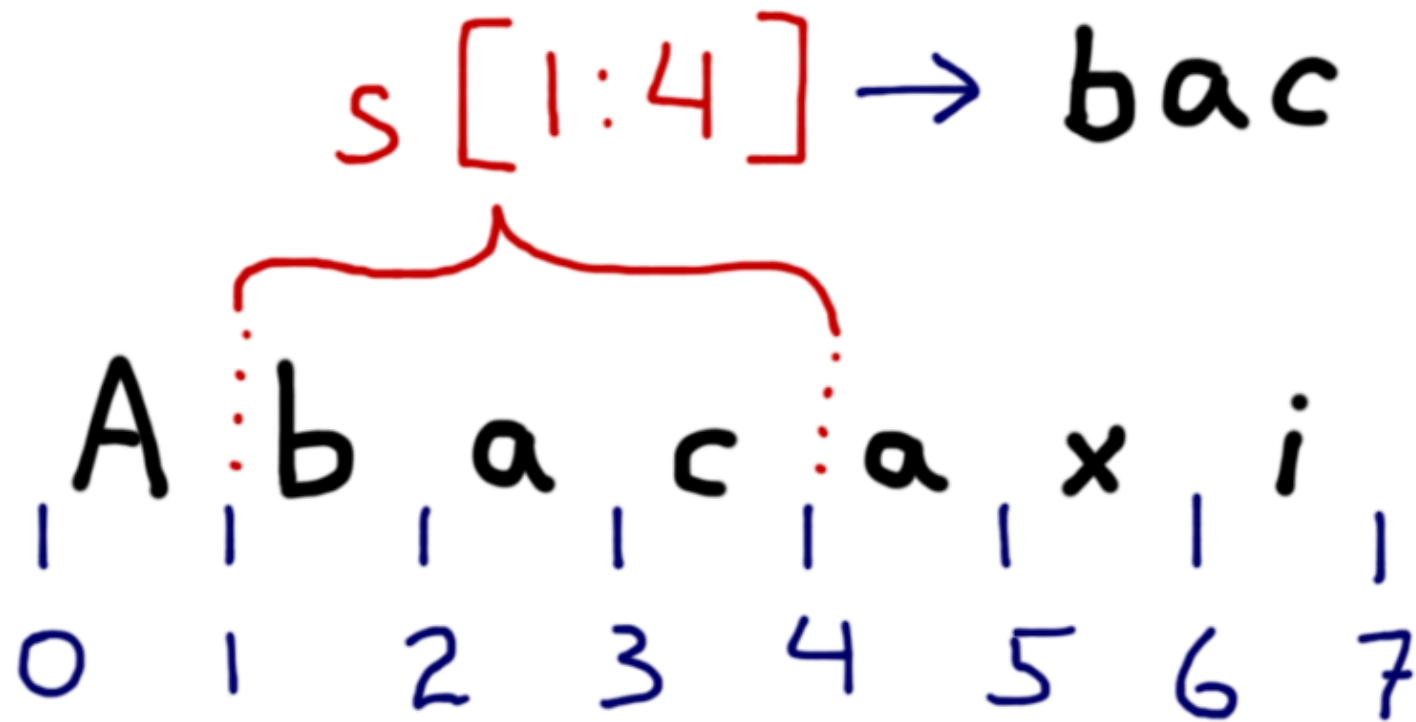
Operações com sequências

- Sequências são coleções ordenadas
 - embutidas: strings, listas, tuplas, buffers
- Operadores:
 - **s[i]** acesso a um item
 - **s[-i]** acesso a um item pelo final
 - **s+z** concatenação
 - **s*n** **n** cópias de **s** concatenadas
 - **i in s** teste de inclusão
 - **i not in s** teste de inclusão negativo

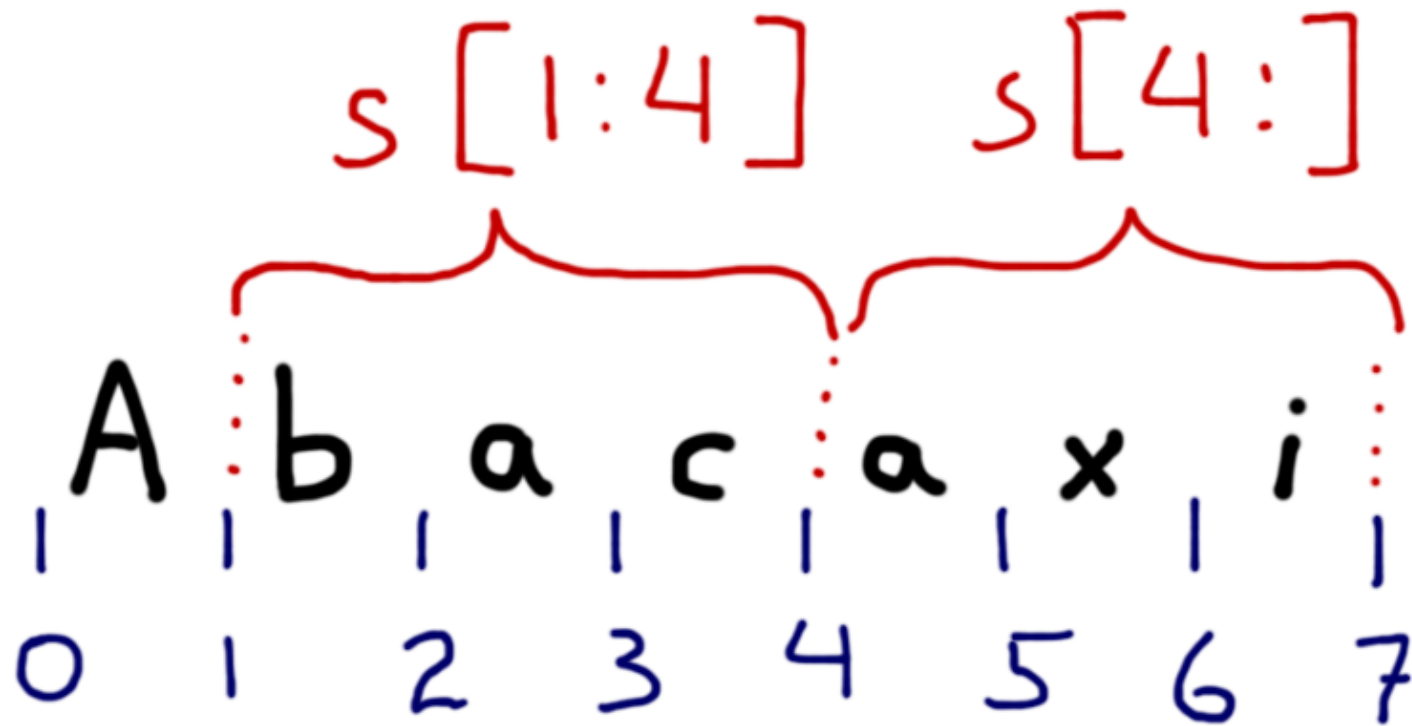
Fatiamento de sequências

- **`s[a:b]`** cópia de **`a`** (inclusive) até **`b`** (exclusive)
- **`s[a:]`** cópia a partir de **`a`** (inclusive)
- **`s[:b]`** cópia até **`b`** (exclusive)
- **`s[:]`** cópia total de **`s`**
- **`s[a:b:n]`** cópia de **`n`** em **`n`** itens
- Atribuição em fatias:
 - **`s[2:5] = [4,3,2,1]`**
 - válida apenas em sequências mutáveis

Abacaxi fatiado

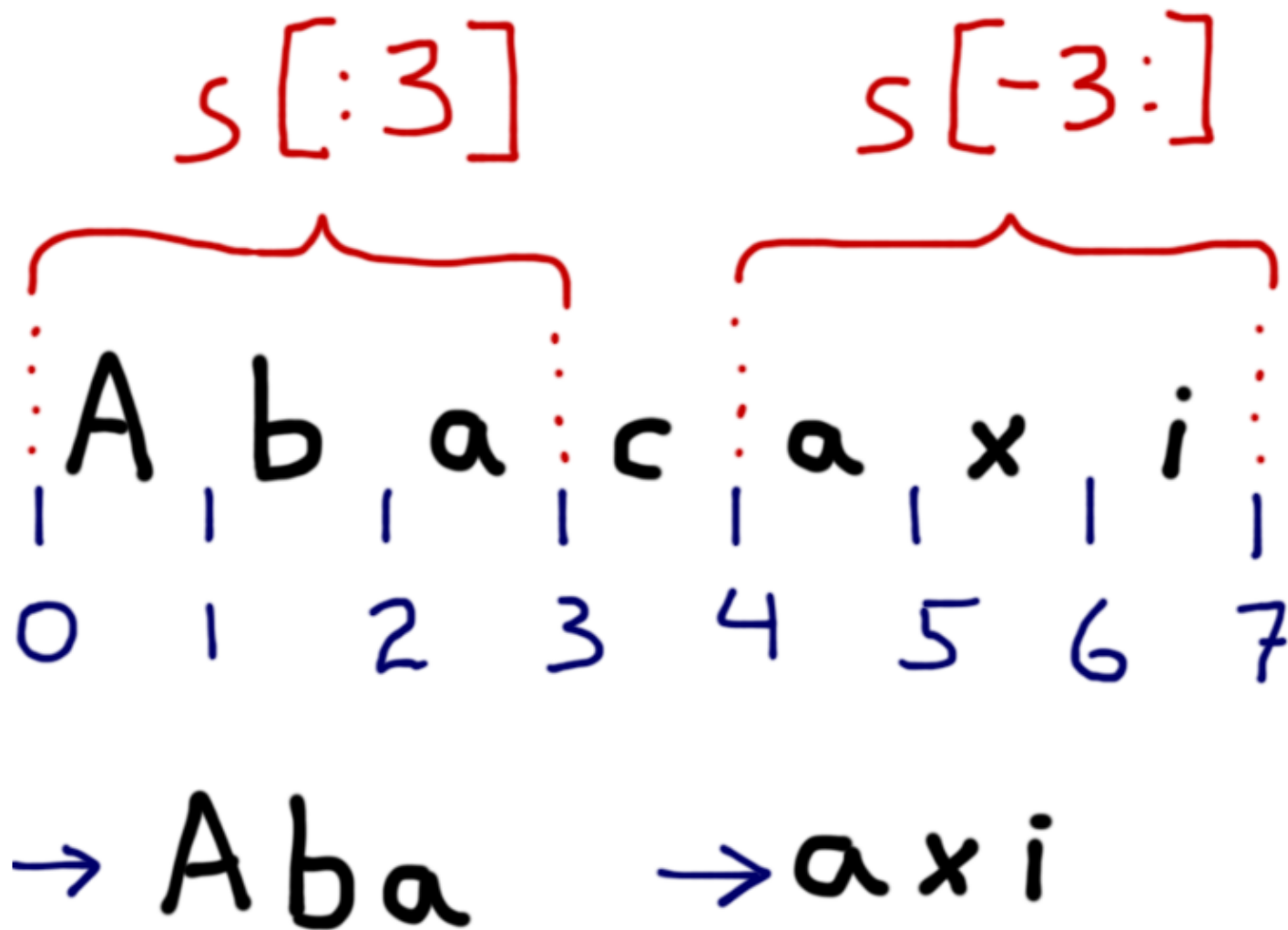


Abacaxi fatiado



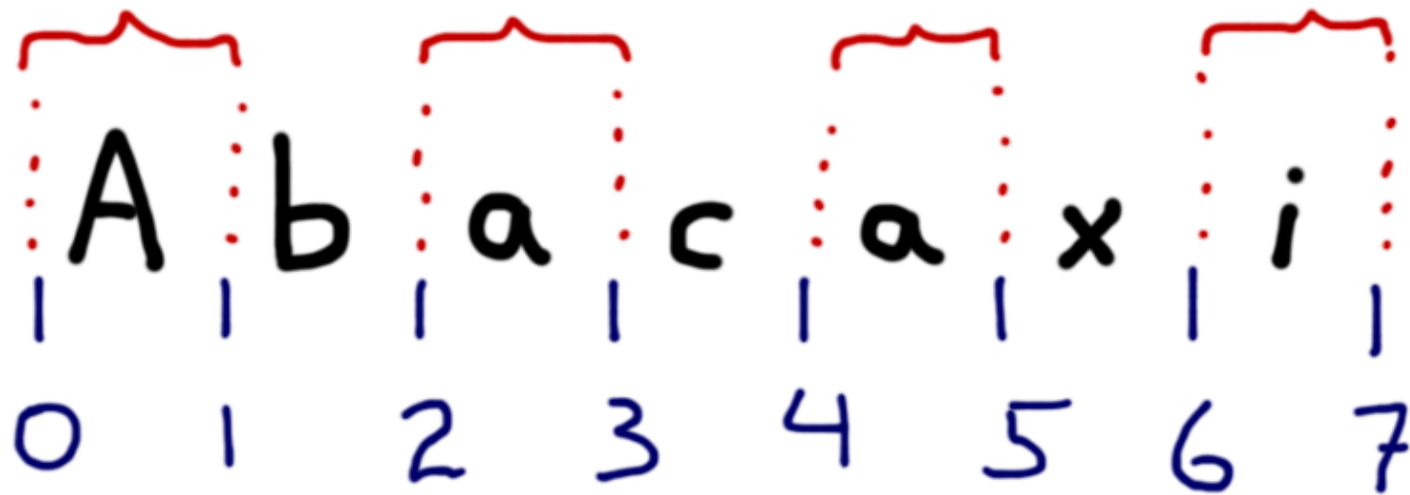
→ axi

Abacaxi fatiado



Abacaxi fatiado

`s[::2]` → Aaai



Funções nativas p/ sequências

- **len(s)**
 - número de elementos
- **min(s), max(s)**
 - valores mínimo e máximo contido em **s**
- **sorted(s)**
 - devolve uma lista com itens de **s** em ordem ascendente
 - `sorted(s, cmp=None, key=None reversed=False)`
- **reversed(s)**
 - retorna um iterador para percorrer os elementos do último ao primeiro

Algumas funções com strings

- **chr(n)**: retorna uma string com um caractere de 8-bits cujo código é **n**
- **unichr(n)**: retorna uma string com um caractere Unicode cujo código é **n**
- **ord(c)**: retorna o código numérico do caractere **c** (pode ser Unicode)
- **repr(x)**: conversão de objeto para sua representação explícita em **Python**
- **len(s)**: número de bytes da string

Alguns métodos de strings

- **s.strip()**
 - retira os brancos (espaços, tabs e newlines) da frente e de trás de **s** (+ parâmetros)
 - **rstrip** e **lstrip** retiram à direita e à esquerda
- **s.upper()**, **s.lower()**, **s.capitalize()**
 - converte todas maiúsculas, todas minúsculas, primeira maiúscula por palavra
- **s.isdigit()**, **s.isalnum()**, **s.islower()**...
 - testa se a string contém somente dígitos, ou somente dígitos e letras ou só minúsculas

Buscando substrings

- **sub in s**
 - **s contém sub?**
- **s.startswith(sub), s.endswith(sub)**
 - **s começa ou termina com sub?**
- **s.find(sub), s.index(sub)**
 - **posição de sub em s (se **sub** não existe em **s**, **find** retorna -1, **index** sinaliza **ValueError**)**
 - **rfind** e **rindex** começam pela direita
- **s.replace(sub1, sub2)**
 - **devolve nova string, substituindo ocorrências de sub1 por sub2**

Aprendendo a aprender

- Use o interpretador interativo!
- Determinar o tipo de um objeto:
 - `type(obj)`
- Ver docs de uma classe ou comando
 - `help(list)`
- Obter uma lista de (quase) todos os atributos de um objeto
 - `dir(list)`
- Listar símbolos do escopo corrente

Como gerar strings com variáveis embutidas

- Operador de interpolação: **f % tupla**

```
>>> m = 'Euro'
>>> t = 2.7383
>>> f = '0 %s está cotado a R$ %0.2f.'
>>> print f % (m,t)
0 Euro está cotado a R$ 2.74.
```

- Tipos de conversão mais comuns:
 - **%s**, **%f**, **%d**: string, float, inteiro decimal
- Aprendendo a aprender:
 - Google: Python String Formatting Operations

Variáveis

- Variáveis contém referências a objetos
 - variáveis **não** “contém” os objetos em si
 - são apenas referências aos objetos
- Variáveis não têm tipo
 - os objetos aos quais elas se referem têm tipo
- Uma variável não pode ser utilizada em uma expressão sem ter sido inicializada
 - não existe “criação automática” de variáveis

Atribuição

- Forma simples
 - `reais = euros * taxa`
- Outras formas
 - atribuição com operação
 - `a+=10` `# a=a+10`
 - atribuição múltipla
 - `x=y=z=0`
 - atribuição desempacotando itens de sequências
 - `a,b,c=lista`
 - `i,j=j,i` `# swap`

Atribuição

- Exemplo

```
# Início da série de Fibonacci  
  
a = b = 1  
while b < 100:  
    print a  
    a, b = b, a + b
```


Execução condicional

- Forma simples
 - **if** cond: comando
- Forma em bloco
 - **if** cond:
 comando1
 comando2
- Alternativas
 - **if** cond1: comando1
 elif cond2: comando 2
 else: comando 3

Repetições: comando **for**

- Para percorrer sequências previamente conhecidas
 - **for** item **in** lista:
 print item
- Se for necessário um índice numérico:
 - **for** idx, item **in** enumerate(lista):
 print idx, item
- Para percorrer uma PA de 0 a 99:
 - **for** i **in** range(100):
 print i

Repetições: comando **while**

- Para repetir enquanto uma condição é verdadeira

```
""" Série de Fibonacci
    até 1.000.000
"""
a = b = 1
while a < 10**6:
    print a
    a, b = b, a + b
```

Controle de repetições

- Para iniciar imediatamente a próxima volta do loop, use o comando **continue**

```
""" Ignorar linhas em branco
"""

soma = 0
for linha in file('vendas.txt'):
    if not linha.strip():
        continue
    codigo, qtd, valor = linha.split()
    soma += qtd * valor
print soma
```

Controle de repetições (2)

- Para encerrar imediatamente o loop, use o comando **break**

```
total=0
while True:
    p = raw_input('+')
    if not p.strip(): break
    total += float(p)
print '-----'
print total
```

Tratamento de exceções

- Comando **try/except**

```
total=0
while True:
    p = raw_input('+')
    if not p.strip(): break
    try:
        total += float(p)
    except ValueError:
        break
print '-----'
print total
```

Palavras reservadas

- **as**
- **and**
- **assert**
- **break**
- **class**
- **continue**
- **def**
- **del**
- **elif**
- **else**
- **except**
- **exec**
- **finally**
- **for**
- **from**
- **global**
- **if**
- **import**
- **in**
- **is**
- **lambda**
- **not**
- **or**
- **pass**
- **print**
- **raise**
- **return**
- **try**
- **while**
- **yield**

Strings unicode

- Padrão universal, compatível com todos os idiomas existentes (português, chinês, grego, híndi, árabe, suaíli etc.)
- Cada caractere é representado por dois bytes
- Utilize o prefixo **u** para denotar uma constante unicode:
u 'maçã'

```
>>> fruta = u'maçã'  
>>> fruta  
u'ma\xe7\xe3'  
>>> print fruta  
maçã  
>>> len(fruta)  
4
```


Conversao entre **str** e **unicode**

- De **str** para **unicode**:
 - `u = s.decode('iso-8859-15')`
- De **unicode** para **str**:
 - `s2 = u.encode('utf-8')`
- O argumento de ambos métodos é uma string especificando a codificação a ser usada

Codificações comuns no Brasil

- **iso-8859-1**: padrão ISO Latin-1
- **iso-8859-15**: idem, com símbolo € (Euro)
- **cp1252**: MS Windows codepage 1252
 - ISO Latin-1 aumentado com caracteres usados em editoração eletrônica (“ ” •)
- **utf-8**: Unicode codificado em 8 bits
 - compatível com ASCII até o código 127
 - utiliza 2 bytes para caracteres não-ASCII
 - este é o padrão recomendado pelo W3C, e é para onde todas os sistemas estão migrando

Codificação em scripts

- As constantes str ou unicode são interpretadas segundo a codificação declarada num comentário especial no início do arquivo .py:

```
#!/usr/bin/env python  
# coding: utf-8
```

```
#!/usr/bin/env python  
# -*- encoding: utf-8 -*-
```

Codificação em scripts (2)

- Exemplo:

```
# -*- coding: iso-8859-15 -*-  
  
euro_iso = '€'  
print '%x' % ord(euro_iso)  
euro_unicode = u'€'  
print '%x' % ord(euro_unicode)
```

- Resultado:

```
a4  
20ac
```

Listas

- Listas são coleções de itens heterogêneos que podem ser acessados sequencialmente ou diretamente através de um índice numérico.
- Constantes do tipo lista são delimitadas por colchetes []
 - `a = []`
 - `b = [1, 10, 7, 5]`
 - `c = ['casa', 43, b, [9, 8, 7], u'coisa']`

Listas

- O método **lista.append(i)** é usado para colocar um novo item **i** na lista.
- O método **lista.extend(l)** inclui todos os itens de **l** no final da lista. O resultado é o mesmo da expressão abaixo, só que mais eficiente pois evita copiar todos os itens da lista:

```
lista += l2
```

- Função embutida **len()** retorna o número de itens da lista:

- `len(a)`, `len(b)`, `len(c)` # 0, 4, ?

Listas

- O método **lista.sort()** ordena os itens de forma ascendente e **lista.reverse()** inverte a ordem dos itens dentro da própria lista, e devolvem None.
- A função embutida **sorted(l)** devolve uma lista com os itens de uma lista ou sequência qualquer ordenados, e **reversed(l)** devolve um iterador para percorrer a sequência em ordem inversa (do último para o primeiro item).

Operações com itens de listas

- Atribuição
 - `lista[5] = 123`
- Outros métodos da classe **list**
 - `lista.insert(posicao, elemento)`
 - `lista.pop()` # +params: ver doc
 - `lista.index(elemento)` # +params: ver doc
 - `lista.remove(elemento)`
- Remoção do item
 - `del lista[3]`

Uma função para gerar listas

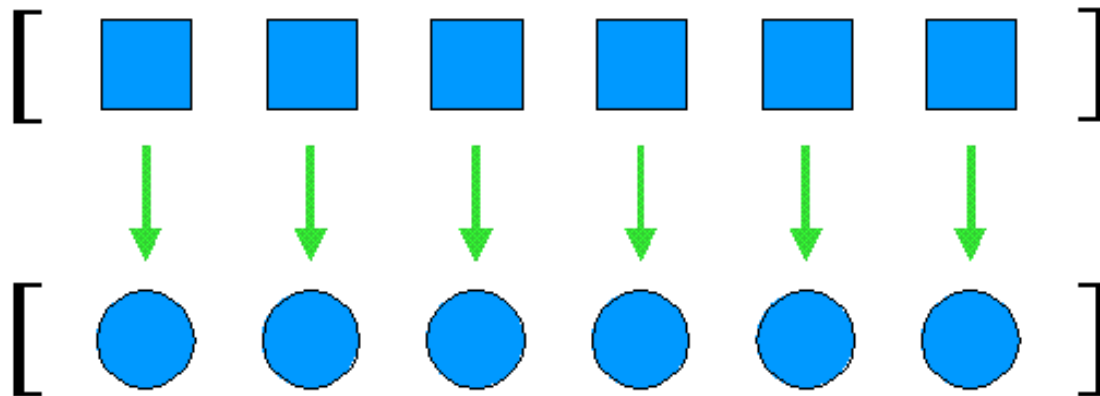
- **range([início,] fim[, passo])**
 - Retorna uma progressão aritmética de acordo com os argumentos fornecidos
- Exemplos:
 - **range(8) # [0, 1, 2, 3, 4, 5, 6, 7]**
 - **range(1, 7) # [1, 2, 3, 4, 5, 6]**
 - **range(1, 8, 3) # [1, 4, 7]**

Expressões para gerar listas

- “List comprehensions” ou “abrangências de listas”
- Produz uma lista a partir de qualquer objeto iterável
- Economizam loops explícitos

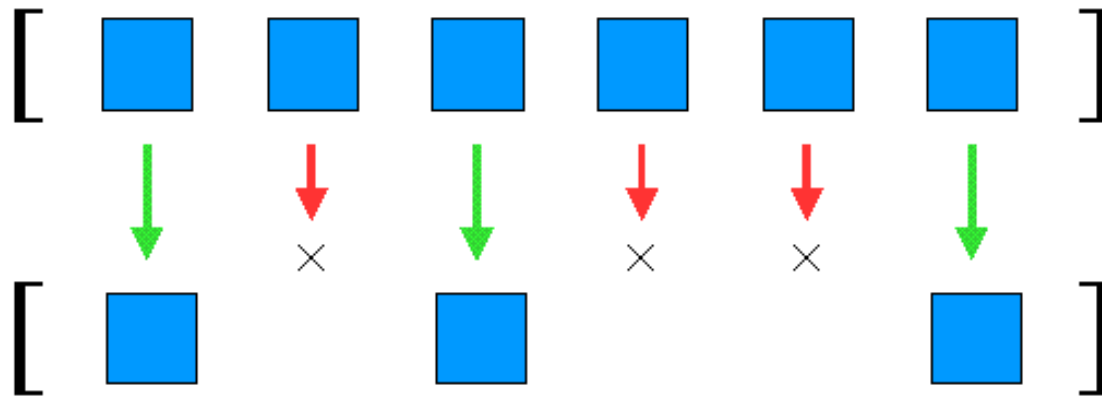
Abrangência de listas

- Sintaxe emprestada da linguagem funcional Haskell
- Processar todos os elementos:
 - $L2 = [n*10 \text{ for } n \text{ in } L]$



Abrangência de listas

- Filtrar alguns elementos:
 - $L2 = [n \text{ for } n \text{ in } L \text{ if } n > 0]$



- Processar e filtrar
 - $L2 = [n*10 \text{ for } n \text{ in } L \text{ if } n > 0]$

Produto cartesiano

- Usando dois ou mais comandos **for** dentro de uma list comprehension

```
>>> qtds = [2,6,12,24]
>>> frutas = ['abacaxis', 'bananas', 'caquis']
>>> [(q,f) for q in qtds for f in frutas]
[(2, 'abacaxis'), (2, 'bananas'), (2, 'caquis'),
 (6, 'abacaxis'), (6, 'bananas'), (6, 'caquis'),
 (12, 'abacaxis'), (12, 'bananas'), (12, 'caquis'),
 (24, 'abacaxis'), (24, 'bananas'), (24, 'caquis')]
```

Produto cartesiano (2)

```
>>> naipes = 'copas ouros espadas paus'.split()
>>> cartas = 'A 2 3 4 5 6 7 8 9 10 J Q K'.split()
>>> baralho = [ (c, n) for n in naipes for c in cartas]
>>> baralho
[('A', 'copas'), ('2', 'copas'), ('3', 'copas'), ('4', 'copas'),
 ('5', 'copas'), ('6', 'copas'), ('7', 'copas'), ('8', 'copas'),
 ('9', 'copas'), ('10', 'copas'), ('J', 'copas'), ('Q', 'copas'),
 ('K', 'copas'), ('A', 'ouros'), ('2', 'ouros'), ('3', 'ouros'),
 ('4', 'ouros'), ('5', 'ouros'), ('6', 'ouros'), ('7', 'ouros'),
 ('8', 'ouros'), ('9', 'ouros'), ('10', 'ouros'), ('J', 'ouros'),
 ('Q', 'ouros'), ('K', 'ouros'), ('A', 'espadas'), ('2', 'espadas'),
 ('3', 'espadas'), ('4', 'espadas'), ('5', 'espadas'),
 ('6', 'espadas'), ('7', 'espadas'), ('8', 'espadas'),
 ('9', 'espadas'), ('10', 'espadas'), ('J', 'espadas'),
 ('Q', 'espadas'), ('K', 'espadas'), ('A', 'paus'), ('2', 'paus'),
 ('3', 'paus'), ('4', 'paus'), ('5', 'paus'), ('6', 'paus'),
 ('7', 'paus'), ('8', 'paus'), ('9', 'paus'), ('10', 'paus'),
 ('J', 'paus'), ('Q', 'paus'), ('K', 'paus')]
>>> len(baralho)
52
```

Tuplas

- Tuplas são sequências imutáveis
 - não é possível modificar as referências contidas na tupla
- Tuplas constantes são representadas como sequências de itens entre parênteses
 - em certos contextos os parênteses em redor das tuplas podem ser omitidos

```
a, b = b, a
```

```
>>> t1 = 1, 3, 5, 7  
>>> t1  
(1, 3, 5, 7)
```

Conversões entre listas e strings

- **s.split([sep[,max]])**

- retorna uma lista de strings, quebrando **s** nos brancos ou no separador fornecido
- **max** limita o número de quebras

- **s.join(l)**

- retorna todas as strings contidas na lista **l** "coladas" com a string **s** (é comum que **s** seja uma string vazia)

```
' '.join(l)
```

- **list(s)**

- retorna **s** como uma lista de caracteres

Tuplas

- Atribuições múltiplas utilizam tuplas

```
#uma lista de duplas
posicoes = [(1,2), (2,2), (5,2), (0,3)]

#um jeito de percorrer
for pos in posicoes:
    i, j = pos
    print i, j

#outro jeito de percorrer
for i, j in posicoes:
    print i, j
```

Dicionários

- Dicionários são coleções de valores identificados por chaves únicas
 - Outra definição: dicionários são coleções de pares chave:valor que podem ser recuperados pela chave
- Dicionários constantes são representados assim:

```
uf={ 'PA' : 'Pará' , 'AM' : 'Amazonas' ,  
     'PR' : 'Paraná' , 'PE' : 'Pernambuco' }
```

Dicionários: características

- As chaves são sempre únicas
- As chaves têm que ser objeto imutáveis
 - números, strings e tuplas são alguns tipos de objetos imutáveis
- Qualquer objeto pode ser um valor
- A ordem de armazenagem das chaves é indefinida
- Dicionários são otimizados para acesso direto a um item pela chave, e não para acesso sequencial em determinada ordem

Dicionários: operações básicas

- Criar um dicionário vazio
 - `d = {}`
 - `d = dict()`
- Acessar um item do dicionário
 - `print d[chave]`
- Adicionar ou sobrescrever um item
 - `d[chave] = valor`
- Remover um item
 - `del d[chave]`

Alguns métodos de dicionários

- Verificar a existência de uma chave
 - `d.has_key(c)`
 - `c in d`
- Obter listas de chaves, valores e pares
 - `d.keys()`
 - `d.values()`
 - `d.items()`
- Acessar um item que talvez não exista
 - `d.get(chave) #res: None ou default`

Conjuntos

- Conjuntos são coleções de itens únicos e imutáveis
- Existem duas classes de conjuntos:
 - **set**: conjuntos mutáveis
 - suportam `s.add(item)` e `s.remove(item)`
 - sintaxe nativa (Python \geq 2.7): `{22, 13, 5, 8}`
 - valores têm que ser objetos imutáveis e *hashable*
 - **frozenset**: conjuntos imutáveis
 - podem ser elementos de outros conjuntos e chaves de dicionários pois são *hashable*

Removendo repetições

- Transformar uma lista num set e depois transformar o set em lista remove todos os itens duplicados da lista

```
l = [2, 6, 6, 4, 4, 6, 1, 4, 2, 2]
s = set(l)
l = list(s)
print l
# [1, 2, 4, 6]
```

Arquivos

- Objetos tipo **file** representam arquivos em disco
- Para abrir um arquivo, use a função **open()**:
 - abrir arquivo binário para leitura
 - `arq = open('/home/juca/grafico.png','rb')`
 - abrir arquivo texto para escrita
 - `arq = open('/home/juca/nomes.txt','w')`
 - abrir arquivo para acrescentar (append)
 - `arq = open('/home/juca/grafico.png','a')`

Definição de funções

- Comando **def** inicia a definição
- Comando **return** marca o fim da execução da função e define o resultado a ser devolvido

```
def inverter(texto):  
    if len(texto) <= 1:  
        return texto  
    lista = list(texto)  
    lista.reverse()  
    return ''.join(lista)
```

Argumentos de funções

- Valores default indicam args. opcionais
 - argumentos obrigatórios devem vir antes de argumentos opcionais

```
def exibir(texto, estilo=None, cor='preto'):
```

- Palavras-chave podem ser usadas para fornecer argumentos fora de ordem
- Como a função acima pode ser invocada:

```
exibir('abacaxi')
```

```
exibir('abacaxi', 'negrito', 'amarelo')
```

```
exibir('abacaxi', cor='azul')
```

Argumentos arbitrários

- Use `*args` para aceitar uma lista de argumentos posicionais
- Use `**args` para aceitar um dicionário de argumentos identificados por palavras-chave
- Exemplo:

```
def tag(nome, *linhas, **atributos):
```

Argumentos arbitrários (2)

```
print tag('br')
print tag('img',src='foto.jpg',width=3,height=4)
print tag('a','Wikipédia',
         href='http://wikipedia.org')
print tag('p','Eu não devia te dizer',
         'mas essa lua','mas esse conhaque',
         'botam a gente comovido como o diabo.',
         id='poesia')
```

```
<br />
```

```

```

```
<a href="http://wikipedia.org">Wikipédia</a>
```

```
<p id="poesia">
```

```
    Eu não devia te dizer
```

```
    mas essa lua
```

```
    mas esse conhaque
```

```
    botam a gente comovido como o diabo.
```

Argumentos arbitrários (3)

```
def tag(nome, *linhas, **atributos):
    saida = ['<' + nome]
    for par in atributos.items():
        saida.append(' %s="%s"' % par)
    if linhas:
        saida.append('>')
        if len(linhas) == 1:
            saida.append(linhas[0])
        else:
            saida.append('\n')
            for linha in linhas:
                saida.append('\t%s\n' % linha)
        saida.append('</%s>' % nome)
    else:
        saida.append(' />')
    return ''.join(saida)
```