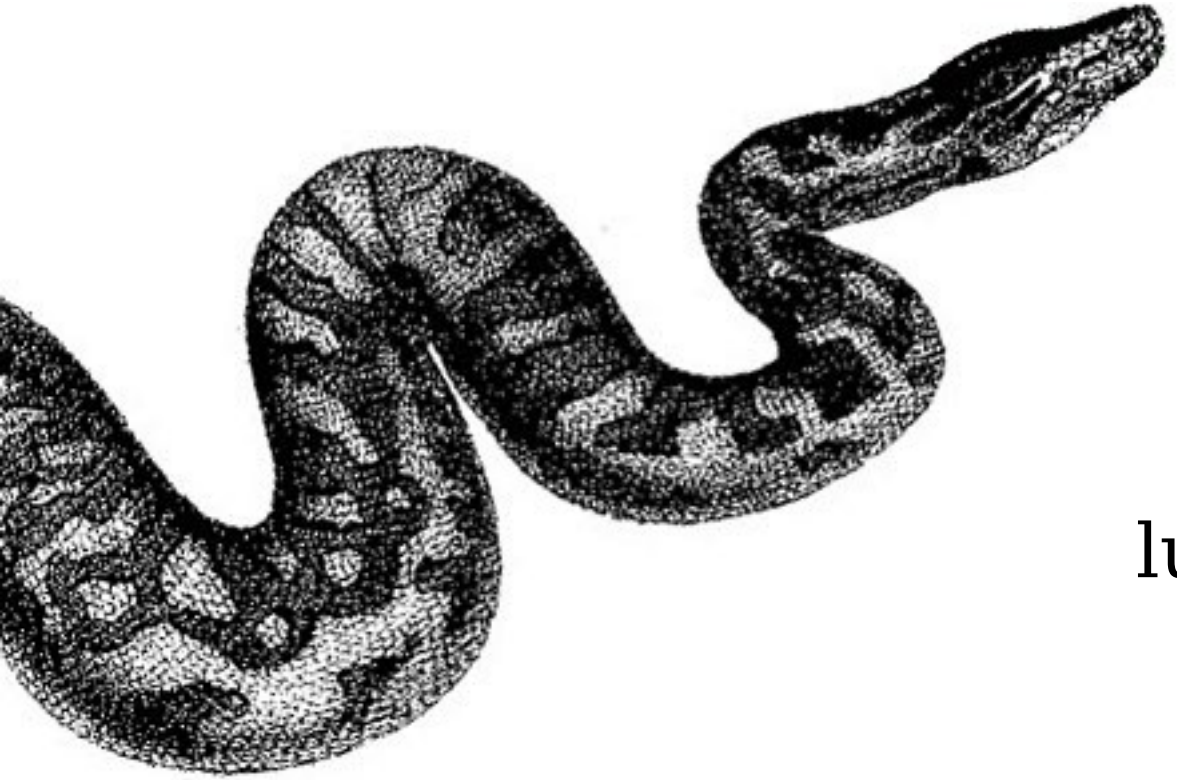
Python:
a primeira mordida



Luciano Ramalho
luciano@occam.com.br

occam

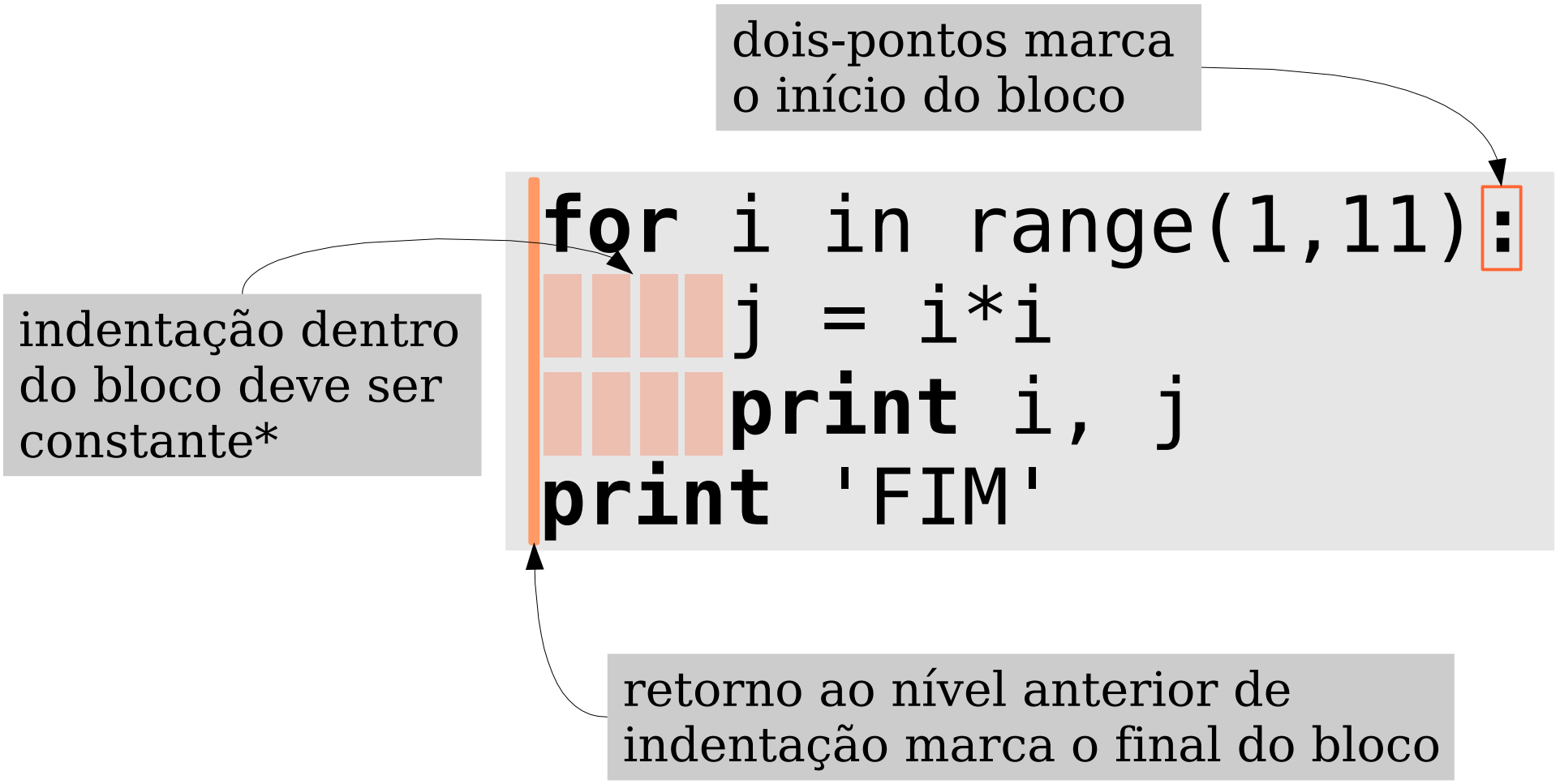
Em vez de *Hello World...*

```
from datetime import datetime
from time import sleep

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

occam

Blocos por endentação



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

occam

Blocos

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

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

occam

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. """
```

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

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

- Python promove de **`int`** para **`long`** automaticamente

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

occam

Outros números

- **`float`**: ponto-flutuante de 32 bits
- **`complex`**: número complexo
- Construtores ou funções de conversão:
  - **`int(a)`**
  - **`long(b)`**
  - **`float(c)`**
  - **`complex(d)`**
  - **`abs(e)`**

```
>>> c = 4 + 3j
>>> abs(c)
5.0
>>> c.real
4.0
>>> c.imag
3.0
```

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() etc.`

Booleanos

- 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
```

occam

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
```

occam

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 especifcando 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, 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
```

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
```

occam

Como gerar strings com variáveis embutidas

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

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

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 caracteres 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)**
  - substitui as 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
  - `dir()`

occam

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

- Função primitiva **`len()`** retorna o número de itens da lista:
  - `len(a), len(b), len(c) # 0, 4, ?`
- O método **`lista.sort()`** ordena os itens de forma ascendente e **`lista.reverse()`** inverte a ordem dos itens dentro da lista.

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]`**

occam

Uma função para gerar listas

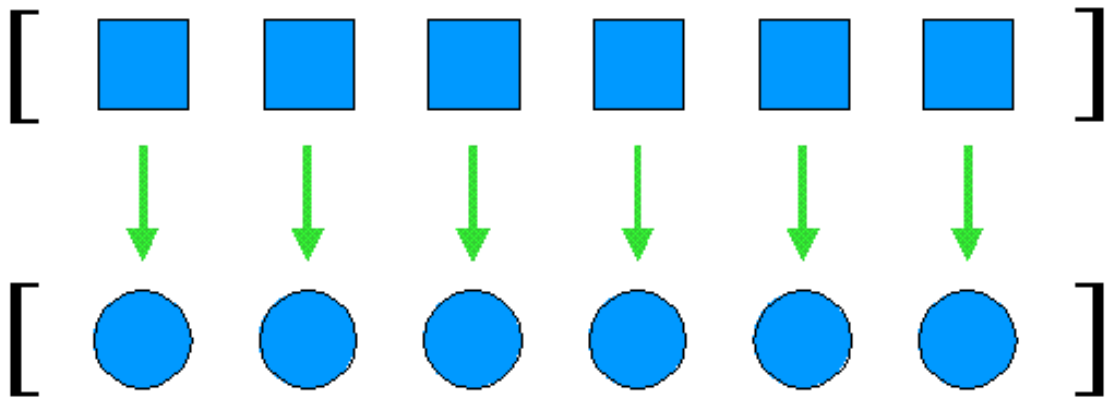
- **range(**[inicio,] fim[, passo]**)**
  - Retorna uma progressão aritmética de acordo com os argumentos fornecidos
- Exemplos:
  - **range(8) # [0,1,2,3,4,5,6,7]**
  - **range(1,7) # [1,2,3,4,5,6]**
  - **range(1,8,3) # [1,4,7]**

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

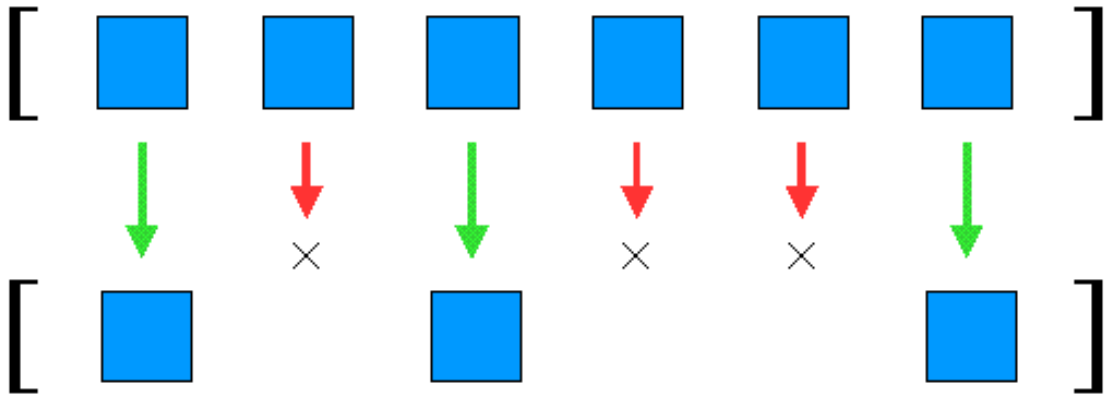
- Sintaxe emprestada da linguagem funcional Haskell
- Processar todos os elementos:
  - L2 = [n*10 for n in L]



occam

Abrangência de listas

- Filtrar alguns elementos:
  - L2 = [n for n in L if n > 0]



- Processar e filtrar
  - L2 = [n*10 for n in L if n > 0]

occam

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 parenteses
  - em certos contextos os parenteses em redor das tuplas podem ser omitidos

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

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

occam

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
```

Operações com sequências

- Sequências são coleções ordenadas
  - nativamente: 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

Funções nativas p/ sequências

- **`len(s)`**
  - número de elementos
- **`min(s), max(s)`**
  - valores mínimo e máximo contido em **`s`**
- **`sorted(s)`**
  - retorna um iterador para percorrer os elementos em ordem ascendente
- **`reversed(s)`**
  - retorna um iterador para percorrer os elementos do último ao primeiro

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) #retorna 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)
  - **frozenset**: conjuntos imutáveis
    - podem ser elementos de outros conjuntos e chaves de dicionários

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 da classe **file** representam arquivos em disco
- Para abrir um arquivo, use o construtor **file()** (a função **open()** é um sinônimo)
  - abrir arquivo binário para leitura
    - arq = file('/home/juca/grafico.png','rb')
  - abrir arquivo texto para escrita
    - arq = file('/home/juca/nomes.txt','w')
  - abrir arquivo para acrescentar (append)
    - arq = file('/home/juca/grafico.png','a')

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
```

occam

Repetições: comando for

- Para percorrer sequências previamente conhecidas
  - ```
    for item in lista:
        print item
    ```
- Sendo 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
```

occam

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
```

occam

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
```

occam

Palavras reservadas

- 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

occam

Variáveis

- Variáveis contém referências a objetos
  - variáveis **não** “contém” os objetos em si
- 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 posicional itens de sequências
    - `a,b,c=lista`
    - `i,j=j,i  # swap`

Atribuição

- Exemplo

```
# Série de Fibonacci

a = b = 1
while True:
    print a
    a, b = b, a + b
```

occam

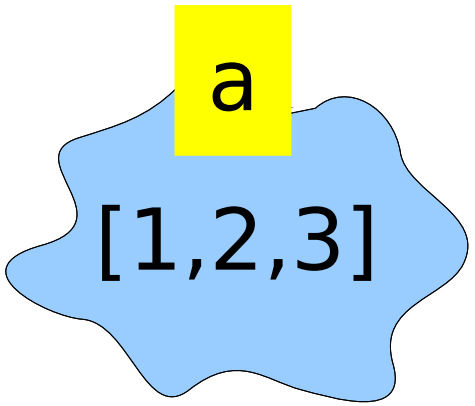
Atribuição: princípios

- Python trabalha com referências, portanto a atribuição não gera uma cópia do objeto
  - Uma variável **não** é uma caixa que contém um valor (esqueça esta velha idéia!)
  - Uma variável é uma etiqueta Post-it colada a um objeto (adote esta nova idéia!!!)
- **del**: comando de desatribuição
  - remove uma referência ao objeto
  - não existindo mais referências, o objeto é varrido da memória

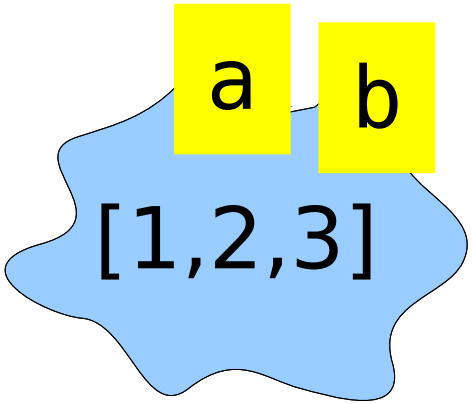
Variáveis

- Podem ser entendidas como rótulos
  - não são "caixas que contém valores"
- Atribuir valor à variável equivale a colar um rótulo no valor

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




```
b = a
```



occam

Apelidos e cópias

```
>>> a = [21, 52, 73]
>>> b = a
>>> c = a[:]
>>> b is a
True
>>> c is a
False
>>> b == a
True
>>> c == a
True
```

- **a** e **b** são apelidos do mesmo objeto lista
- **c** é uma referência a uma cópia da lista

```
>>> a, b, c
([21, 52, 73], [21, 52, 73], [21, 52, 73])
>>> b[1] = 999
>>> a, b, c
([21, 999, 73], [21, 999, 73], [21, 52, 73])
>>>
```

Definição de funções

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

```
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 />
<img src="foto.jpg" height="4" width="3" />
<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.
</p>
```

occam

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)
```

occam

Classes

```
class Contador(object):
    def __init__(self):
        self.dic = {}

    def incluir(self, item):
        qtd = self.dic.get(item, 0) + 1
        self.dic[item] = qtd
        return qtd

    def contar(self, item):
        return self.dic[item]
```

- declaração
- inicializador
- métodos

occam

Classes “vazias”

- Estilo antigo (old style)
  - **pass** indica um bloco vazio

```
class Coisa:
    pass
```

- Estilo novo (new style)

```
class Coisa(object):
    pass
```

- É possível definir atributos nas instâncias

```
>>> c = Coisa()
>>> c.altura = 2.5
>>> c.largura = 3
>>> c.altura, c.largura
(2.5, 3)
>>> dir(c)
['__doc__', '__module__', 'altura', 'largura']
```

Como extender uma classe

```
class ContadorTolerante(Contador):

    def contar(self, item):
        return self.dic.get(item, 0)
```

- declaração
- método sobrescrito

- Como invocar métodos de super-classes:

```
class ContadorTotalizador(Contador):

    def __init__(self):
        super(ContadorTotalizador, self).__init__()
        self.total = 0

    def incluir(self, item):
        super(ContadorTotalizador, self).incluir(item)
        self.total += 1
```

Herança múltipla

```
class ContadorTolerante(Contador):
    def contar(self, item):
        return self.dic.get(item, 0)

class ContadorTotalizador(Contador):
    def __init__(self):
        super(ContadorTotalizador, self).__init__()
        self.total = 0

    def incluir(self, item):
        super(ContadorTotalizador, self).incluir(item)
        self.total += 1

class ContadorTT(ContadorTotalizador,ContadorTolerante):
    pass
```

- **pass** indica um bloco vazio

occam

Propriedades

- Encapsulamento quando você precisa

```
>>> a = C()
>>> a.x = 10        #!!!
>>> print a.x
10
>>> a.x = -10
>>> print a.x       # ??????
0
```

occam

Implementação de uma propriedade

- Apenas para leitura

```
class C(object):
    def __init__(self, x):
        self.__x = x
    @property
    def x(self):
        return self.__x
```

```
class C(object):
    def __init__(self, x=0):
        self.__x = x
    def getx(self):
        return self.__x
    def setx(self, x):
        if x < 0: x = 0
        self.__x = x
    x = property(getx, setx)
```

- Para leitura e escrita

occam

Exemplo de propriedade

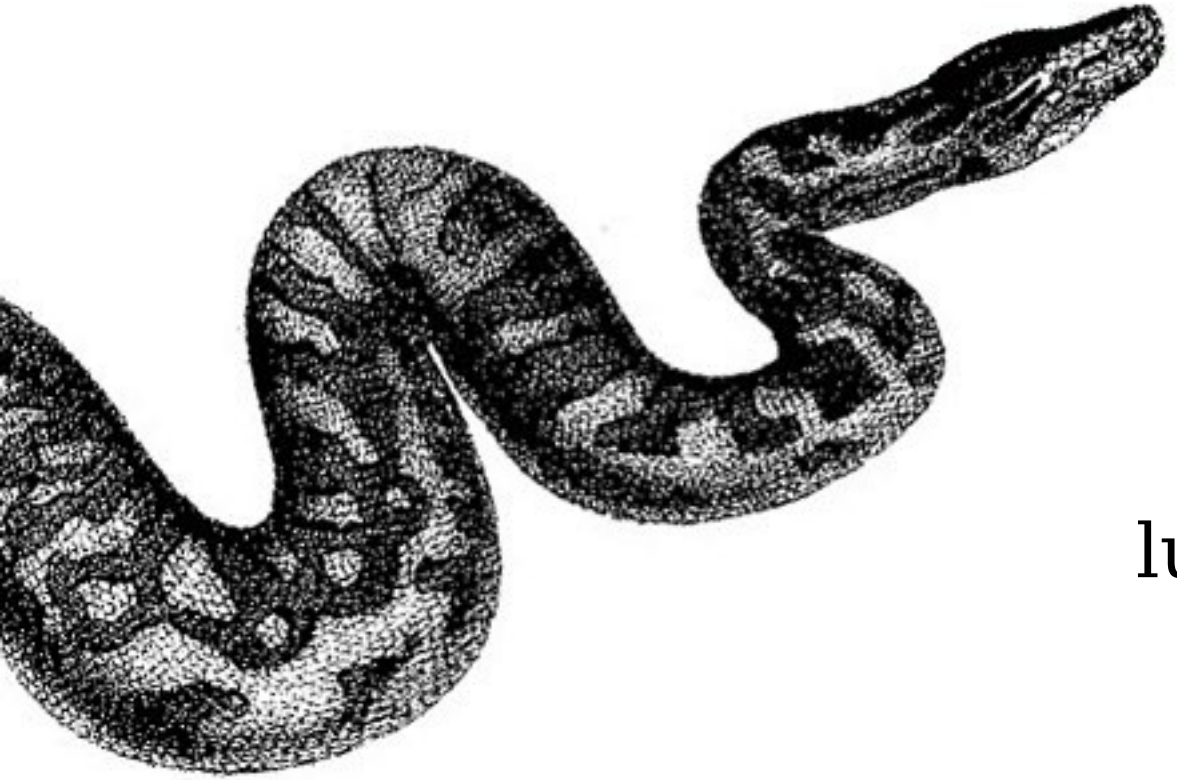
```
class ContadorTotalizador(Contador):
    def __init__(self):
        super(ContadorTotalizador, self).__init__()
        self.__total = 0

    def incluir(self, item):
        super(ContadorTotalizador, self).incluir(item)
        self.__total += 1

    @property
    def total(self):
      return self.__total
```

- Funciona porque é uma classe estilo novo
  - extende de `Contador`, que extende `object`

python.mordida[0]



Luciano Ramalho
luciano@occam.com.br

occam