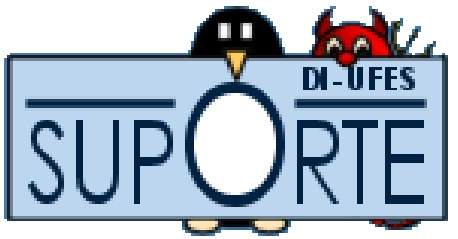
Curso Básico de

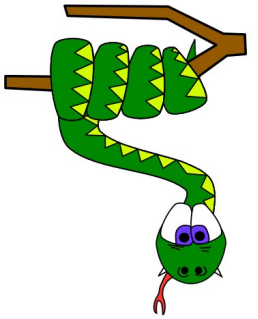
python™

Leonardo Arruda do Amaral Andrade

VII Escola Regional de Informática da SBC
III Encontro de Software Livre no Espírito Santo (ESLES)



Sumário

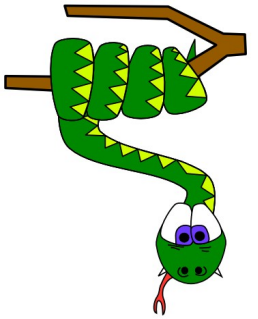


- **Parte 0**
  - Introdução
    - Características gerais
    - Algumas aplicações escritas em Python
    - Onde Python é utilizado?
    - Quem usa Python?

- **Parte 1** (focado em aspectos gerais da linguagem e programação procedural)
  - O interpretador Python
  - Manipulando variáveis
  - Palavras reservadas
  - Alguns tipos básicos da linguagem
  - Operadores booleanos e lógicos
  - Strings
  - Código-fonte de verdade
  - Unicode
  - Estruturas de controle (if, for, while)
  - Estruturas de dados de alto nível (listas, tuplas, dicionários)
  - Funções

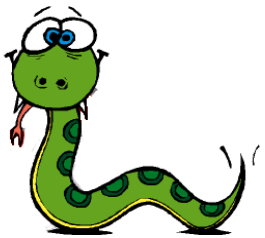
python™

Sumário



- **Parte 1** (focado em aspectos gerais da linguagem e programação procedural)
  - Módulos
  - Arquivos

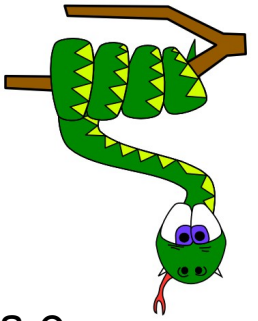
- **Parte 2** (focado em aspectos de orientação a objetos [OO])
  - Classes e objetos
  - Métodos e atributos
  - Herança e herança múltipla
  - Propriedades, classe base e herança
  - Atributos e métodos estáticos
  - Métodos especiais e sobrecarga
  - Mecanismo de exceções



python™

Parte 0

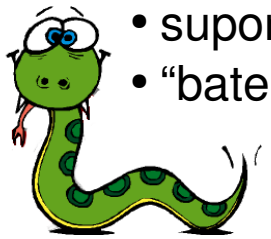
Introdução



Criada por Guido van Rossum (1990-1991), no Instituto Nacional de Matemática e Ciência da Computação (CWI) – Amsterdã, Holanda.

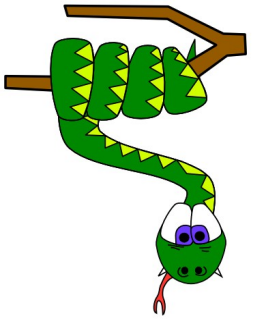
Características gerais:

- bastante clara, sintaxe legível, fácil aprendizagem
- interpretada e interativa
- suporte a vários paradigmas: funcional, procedural e orientado a objetos
- multiplataforma (Windows, Linux/Unix, Mac OS X, OS/2, Amiga, Palm Handhelds e celulares Nokia)
- tipagem dinâmica (não é necessário declarar tipos)
- compilação em tempo de execução (on the fly)
- fortemente tipada
- estruturas de alto nível [listas, tuplas, dicionários, etc] já disponibilizadas pela linguagem
- orientada a objetos
- suporte a módulos e pacotes
- “baterias inclusas” (ampla biblioteca padrão)



python™

Introdução

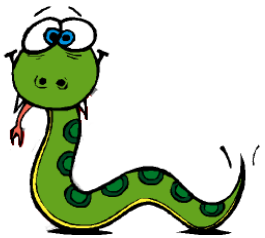


Características gerais:

• faz o programador se concentrar na resolução do problema (questões como compilação, declaração de variáveis, sintaxe complexa, etc).
• possibilidade de integração com código C, C++, Java (Jython), etc.

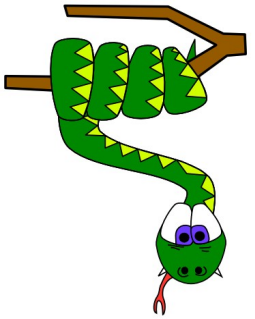
• Ótima linguagem para prototipagem, RAD (Rapid Application Development), extensão e integração de aplicações e componentes, scripts.

Algumas aplicações escritas em Python:
• GmailFS
• Luma LDAP Browser
• PyMol
• Python-uno (OOffice)



python™

Introdução

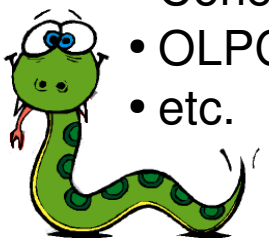


Onde Python é utilizado?

Programação Web (CGI, Zope, Django, TurboGears, XML), Banco de dados (ODBD, MySQL, PyGresQL, etc.), educação, jogos (Pygames), redes, computação gráfica, bioinformática, física, aplicações numéricas, etc, etc, e mais etc.

Quem é doido de usar Python?

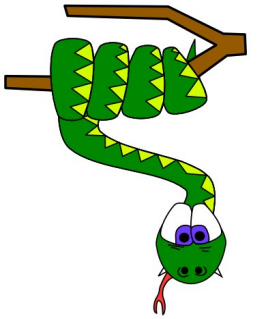
- Philips: controle de produção de microcontroladores
- ILM: integração de computadores e aplicações usados em computação gráfica
- Nokia: aplicações para os seus celulares
- Embratel: monitoramento de interfaces do backbone
- Serpro: desenvolvimento de aplicações Web
- Google
- RedHat
- NASA
- Conectiva
- OLPC
- etc.



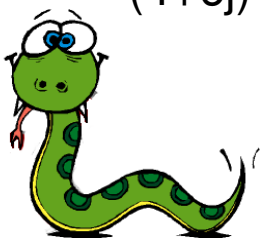
python™

Parte 1

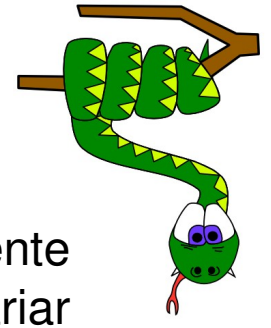
O interpretador Python



```
# Python como uma calculadora
>>> 3 + 4
7
>>> 2 - 2 #Comentario em Python
0
>>> #divisao inteira eh arrendondada para baixo
... 7/3
2
>>> -7/3
-3
>>>
>>> 2 == 2
True
>>> print 'Escreve Python!'
Escreve Python!
>>> (1+2j)+(3+4j)
(4+6j)
```



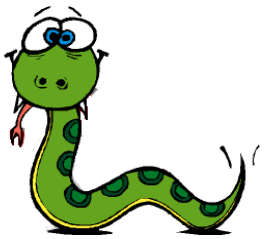
python™



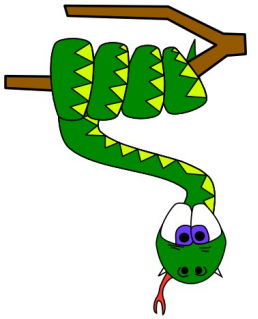
Manipulando variáveis

Em Python, não se declara tipo de variáveis. Os tipos são inferidos dinamicamente (tipagem dinâmica), ou seja, o tipo ao qual a variável está associada pode variar durante a execução do programa. Não quer dizer que não exista tipo específico definido (tipagem fraca). As variáveis assumem um único tipo em um determinado momento.

```
>>> umInteiro=42
>>> type(umInteiro) #type() é uma função build-in
<type 'int'>
>>> umFloat=43.01
>>> umLong=5L
>>> a = b = c = 0
>>> umInteiro * a
0
>>> umInteiro + umFloat
85.009999999999991
```



python™



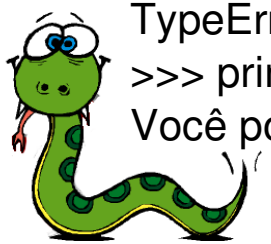
Palavras reservadas

Lista das palavras reservadas de Python:

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

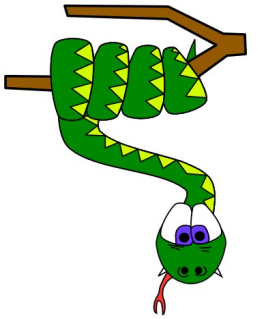
Essas palavras não podem ser utilizadas como nome de variáveis. Se você for teimoso, receberá um erro de sintaxe!
Ah, existem palavras que aparentemente são reservadas mas não são! Portanto podem ser sobrescritas na região de código. Esteja consciente ao fazer isso:

```
>>> str(1)
'1'
>>> str = "Você pode me chamar de louco, mas eu sei o que to fazendo!"
>>> str(1) #você acaba de perder a referência a função str!
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: 'str' object is not callable
>>> print str
Você pode me chamar de louco, mas eu sei o que to fazendo!
```

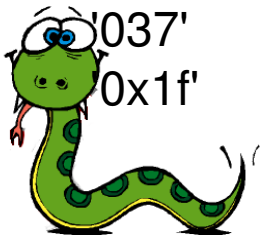


python™

Alguns tipos básicos da linguagem

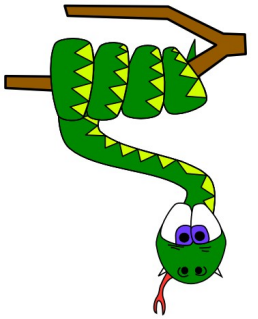


```
>>> a = 31;
>>> a + 1; a - 2; a * 2; a / 2; a // 2; a % 2; a**4; #operações com inteiros
32
29
62
15
15
1
923521
>>> #as operações acima valem para float
>>> a = 31.0;
>>> a / 2; a // 2; #diferenciando os dois operadores
15.5
15.0
>>>
>>> abs(-31); divmod(31,2); oct(31); hex(31); #build-in's
31
(15, 1)
'037'
'0x1f'
```

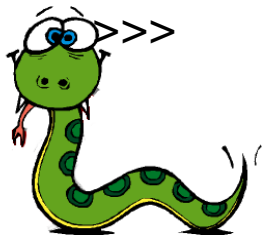


python™

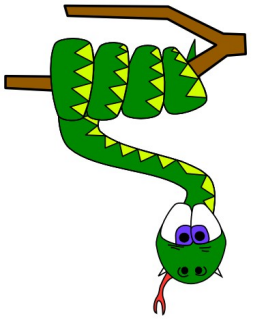
Alguns tipos básicos da linguagem



```
>>> from math import *
>>> cos(0); sin(0); exp(0); log(10); log10(10); pi; e; pow(2,1/2); sqrt(2)
1.0
0.0
1.0
2.302585092994045
1.0
3.141592653589793
2.718281828459045
1.0 #[porquê?]
1.4142135623730951
>>>
>>> #para números complexos, utiliza-se a biblioteca cmath
>>> from cmath import *
>>> exp((complex(1,1)))
(1.4686939399158851+2.2873552871788423j)
>>> exp(1+0j)
(2.718281828459045+0j)
>>>
```



python™



Operadores booleanos e lógicos

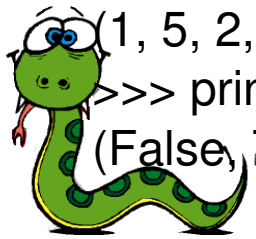
Operadores booleanos

```
>>> 5 == 5; 5 == 6; 5 != 7; 5 > 7; #até aqui como C!
True
False
True
False
>>> 3 < 5 < 8 #Python é capaz de avaliar expressões como essa!
True
```

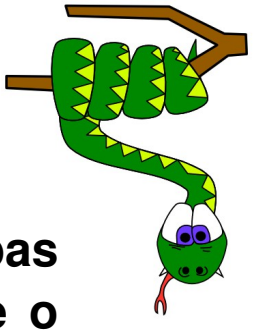
>>> #Valores podem ser convertidos para booleanos através de bool(valor). Normalmente, será 'False' se o valor representar algo “vazio” dentro do seu domínio. Ex.: o número 0, uma lista vazia, uma string vazia, uma tupla vazia, None, etc.

>>> **#Operadores lógicos**

```
>>> print (1 and 0, 5 and 0, 2 and 3, 3 and 2) #se V, recebe o segundo valor!
(0, 0, 3, 2)
>>> print (1 or 0, 5 or 0, 2 or 3, 3 or 2, 0 or 0) #se V, recebe o primeiro valor!
(1, 5, 2, 3, 0)
>>> print (not 1, not 0, not True, not False, not "abacate", not "")
(False, True, False, True, False, True)
```



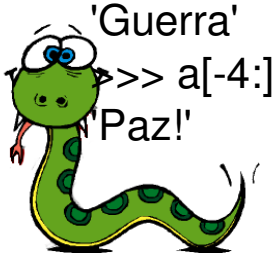
python™



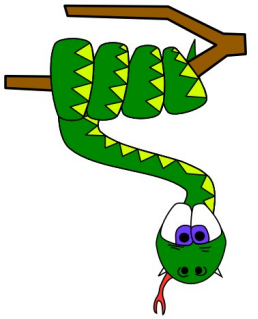
Strings!

As strings são cadeias de caracteres que podem ser delimitadas por aspas simples ou duplas, com igual efeito. Importante salientar que não existe o tipo char. Uma cadeia com um único caractere é visto como string pela linguagem.

```
>>> a = "Guerra"
>>> a = a + "e Paz"
>>> a += '!'
>>> print a
Guerra e Paz!
>>> len(a) #tamanho de uma string
13
>>> a * 3
'Guerra e Paz!Guerra e Paz!Guerra e Paz!'
>>> a[0:6] #slicing (fatiando strings)
'Guerra'
>>> a[6:]
' e Paz!'
>>> a[:6]
'Guerra'
>>> a[-4:]
'Paz!'
```



python™

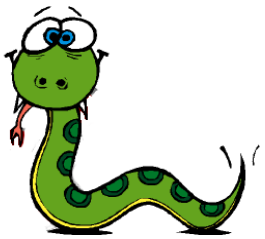


Strings!

```
>>> a[20] #erro ao acessar posição existente!
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
>>>
```

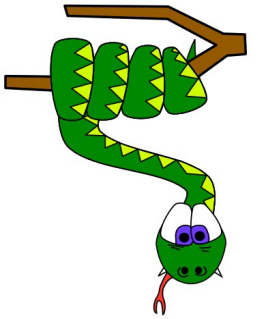
Strings são imutáveis! Isso significa, de modo prático, que não é possível acessar e modificar diretamente um de seus caracteres. Existem também vários métodos relacionados ao tipo string [str] que podem ser vistos através de 'help(str)'.

```
>>> a[-4]
'P'
>>> a[-4] = 'F' #strings são imutáveis
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
```



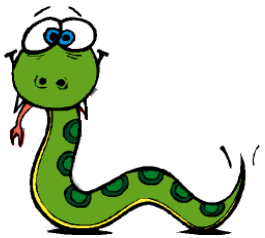
python™

Strings!

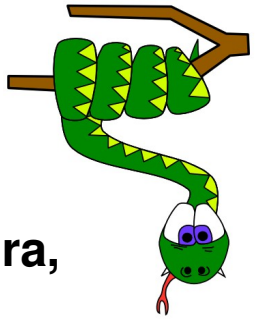


Alguns métodos:

```
>>> a.lower()
'guerra e paz!'
>>> a.upper()
'GUERRA E PAZ!'
>>> a.split()
['Guerra', 'e', 'Paz!']
>>>
>>> ord('a')
97
>>> help(str) #divirta-se!
```



python™



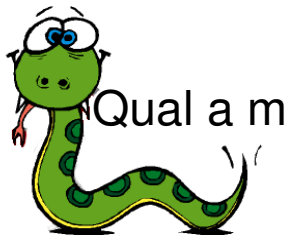
Código-fonte de verdade

Até aqui, utilizamos o interpretador na sua forma interativa. Agora, escreveremos nossos programas em arquivos.

Algumas pequenas observações:

-> Os arquivos fontes de Python, por convenção, possuem extensão '.py'
-> Na família Unix, utiliza-se na primeira linha a localização do interpretador Python. (não é obrigatória)
-> Na segunda linha, informa-se ao interpretador qual a codificação de caracteres usada no código. Se não informada, o interpretador assume que o código foi escrito utilizando-se os caracteres da tabela ASCii.

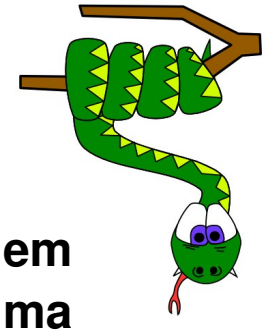
```
#!/usr/bin/python
# -*- coding: iso-8859-1 -*-

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



Qual a mágica por detrás dos panos?

python™

O Unicode

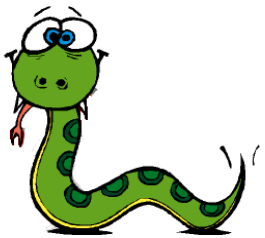


Ao informar a codificação, permite-se que as strings nativas (escritas em ASCII no código fonte) sejam traduzidas pelo programa para uma representação Unicode de forma automática.

Unicode é um tipo em Python. Suas representações:

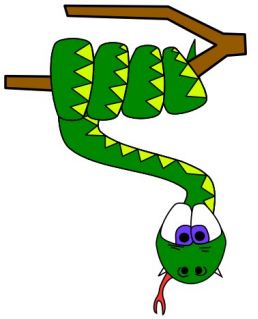
u"Vous pouvez répéter, s'il vous plaît?"
U"Vous pouvez répéter, s'il vous plaît?"
u"""Vous pouvez répéter, s'il vous plaît?"""
U"""Vous pouvez répéter, s'il vous plaît?"""

Basicamente suporta os mesmos métodos de string [str].



python™

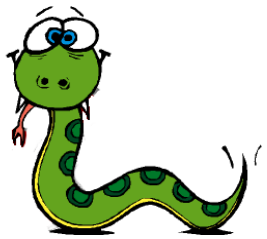
Estruturas de controle



Condicional 'if'

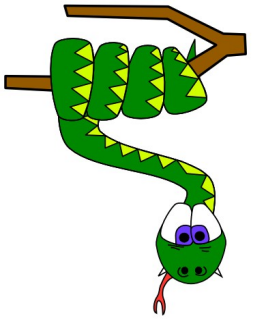
```
if condição:
    #bloco de código
elif condição:
    #outro bloco
else:
    #bloco final
```

identação correta é necessária! Ela delimita os blocos!
Não se admite atribuições como condições do if.
elif não é obrigatória mas é útil (por exemplo, menus). Substitui a construção switch/case de C.



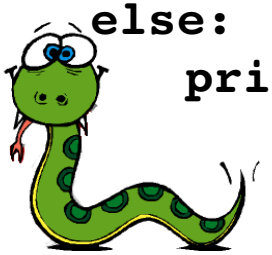
python™

Estruturas de controle



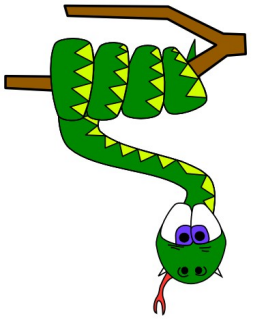
Condicional 'if' (exemplo)

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

print """
    Restaurante do Zequinha
    Manda quem pode, obedece quem tem juízo!
"""

pedido=raw_input("O que deseja almoçar?\n")

if pedido == "arroz":
    print "Sim, senhor. Temos arroz, são R$ 5!"
elif pedido == "feijão":
    print "Sim, senhor. Temos feijão, R$ 10!"
elif pedido == "panquecas":
    print "As panquecas estão frias. São R$ 3 cada!"
else:
    print "Só temos arroz, feijão e panquecas!"
```



python™

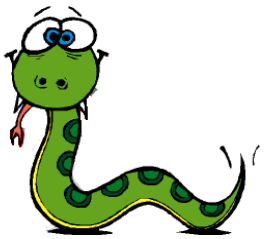
Estruturas de controle



Laço 'for'

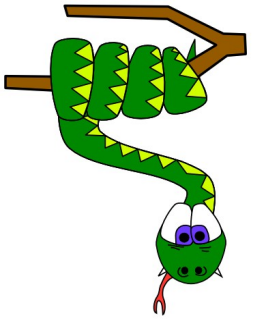
```
for variavel in sequência:
        #bloco de código
        <break>|<continue>
else:
        # bloco executado na ausência de um break
```

Sintaxe bastante diferente de C, próxima a de linguagens de script.
As seqüências normalmente são uma listas, tuplas ou dicionários.
para se fazer um laço com número fixo de iterações, tal como em C, utiliza-se for em conjunto com a função range (que gera seqüências de números).



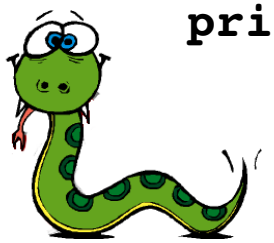
python™

Estruturas de controle



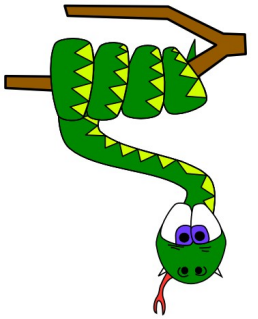
Laço 'for' (exemplo)

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

print """
    Venda do Marquito
"""
print "Produtos:"
lista = ["café", "leite", "macarrão", "fubá"]

for produto in lista:
    print "-->" + produto

print "\nMarquito conta até cinco.\n"
for i in range(1,6):
    print "-->" + str(i)
else:
    print "Marquito é um cara esperto!"
```



python™

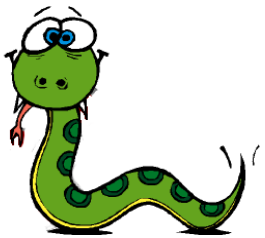
Estruturas de controle



 # Laço 'while'

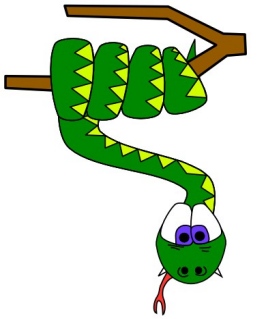
```
while condição:
        #bloco de código
        <break>|<continue>
else:
        # bloco executado na ausência de um break
```

Sem surpresas!



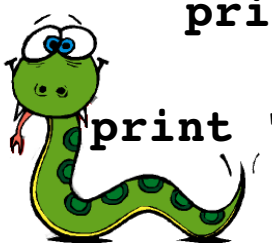
python™

Estruturas de controle



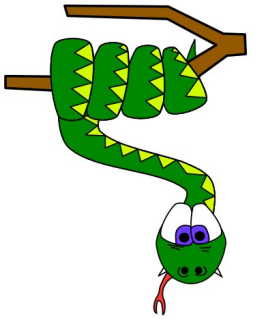
Laço 'while' (exemplo)

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

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

print """
    Espera!
"""
paciencia=10
while (paciencia > 0):
    print "Toh calmo!"
    paciencia -= 1
    if (paciencia == 5):
        break;
else:
    print "Nao entro aqui!"

print "Vai me atender ou nao vai?"
```



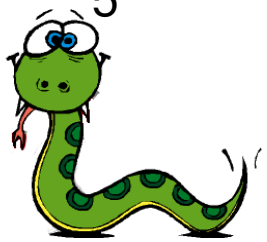
python™

Estruturas de dados de alto nível



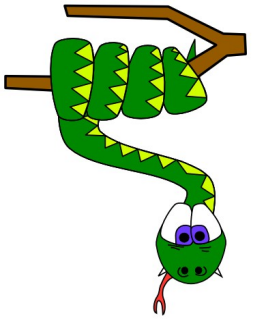
Listas

```
>>> listamista = [1, 1.4, 1+1j, "Qualquer tipo", ["lista dentro de lista"]] #a lista é polimórfica (comportam praticamente todos os tipos de objetos)
>>> print listamista
[1, 1.3999999999999999, (1+1j), 'Qualquer tipo', ['lista dentro de lista']]
>>> listamista[0] #indexável
1
>>> listamista[1:3] #fatiamento
[1.3999999999999999, (1+1j)]
>>> listamista[3]="Entendi!" #mutável
>>> print listamista
[1, 1.3999999999999999, (1+1j), 'Entendi!', ['lista dentro de lista']]
>>>
>>> [] #lista vazia
[]
>>> len(listamista)
5
```



python™

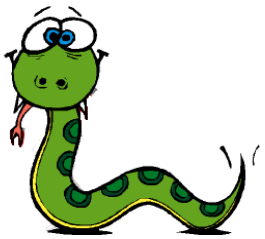
Estruturas de dados de alto nível



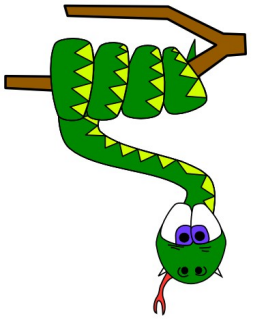
Listas

```
>>> range(1,5)
[1, 2, 3, 4]
>>> matrix = [[1,2,3],[4,5,6],[7,8,9]] #construção de matrizes com listas
>>> matrix[0][0];matrix[1][1];matrix[2][2]
1
5
9
```

Com as listas providas pela linguagem, é possível criar também outras estruturas de dados como filas, pilhas, etc. utilizando-se os métodos do tipo list.



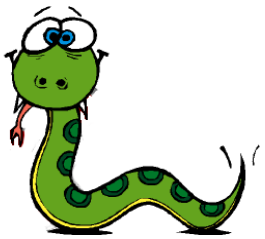
python™



Estruturas de dados de alto nível

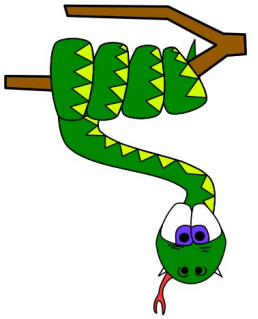
Listas

```
>>> lista = [15,10,5,30,25,20]
>>> lista.append(35); lista #adicionando um elemento ao final da lista
[15, 10, 5, 30, 25, 20, 35]
>>> lista.reverse(); lista #'invertendo' a lista
[35, 20, 25, 30, 5, 10, 15]
>>> lista.pop(); lista #retirando o último elemento da lista
15
[35, 20, 25, 30, 5, 10]
>>> lista.remove(5); lista #retirando o número 5 da lista
[35, 20, 25, 30, 10]
>>> lista.sort(); lista #ordenando a lista
[10, 20, 25, 30, 35]
```



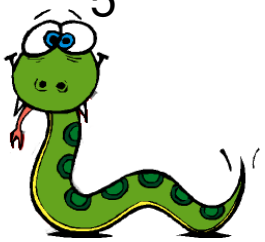
python™

Estruturas de dados de alto nível

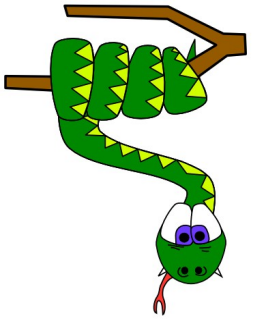


Tuplas

```
>>> tuplamista = (1, 1.4, 1+1j, "Qualquer tipo", ("tupla dentro de tupla")) #tuplas também são polimórficas
>>> print tuplamista
(1, 1.399999999999999, (1+1j), 'Qualquer tipo', 'tupla dentro de tupla')
>>> tuplamista[0] #indexável
1
>>> tuplamista[1:3] #fatiamento
(1.399999999999999, (1+1j))
>>> tuplamista[3]="Erro na cara!" #imutável
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
>>> () #tupla vazia
()
>>> len(tuplamista)
5
```



python™



Estruturas de dados de alto nível

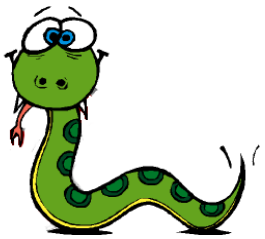
Tuplas

Onde é usada?

Especial importância quando se precisa de uma estrutura imutável. É utilizada para retorno de múltiplos valores de uma função.

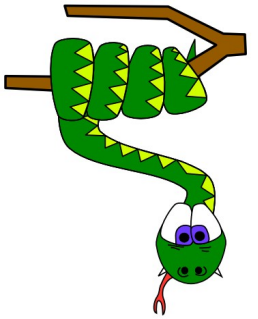
>>> help(tuple)

Não há nenhum método comum (como append, sort, etc.) para o tipo tupla.



python™

Estruturas de dados de alto nível

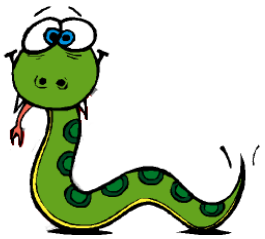


Dicionários (Mapeamentos / Tabelas Hash)

```
>>> dict = {"chave":"valor"}
>>> dicionario = {"Iraque":"Asia", "Brasil":"América", "Espanha":"Europa", "Austrália":"Oceania", "Congo":"África"}
>>> print dicionario["Brasil"]
América
```

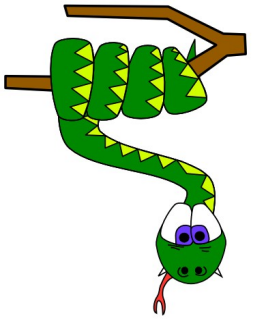
Diferentemente das listas e tuplas (que são indexados por inteiros) são indexados por chaves (keys). Essas chaves podem ser apenas de tipos imutáveis (inteiros, strings, tuplas*, ...).

Dicionário pode ser visto como um conjunto não ordenado de pares chaves-valor, onde as chaves são únicas para uma estrutura dicionário.



python™

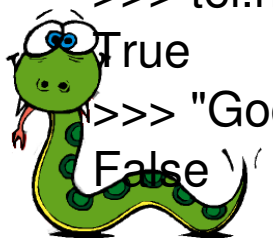
Estruturas de dados de alto nível



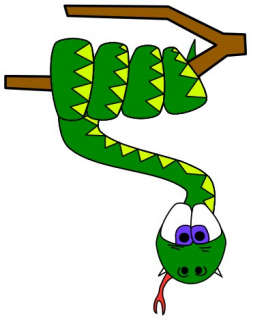
 # Dicionários (Mapeamentos / Tabelas Hash)

As principais operações em um dicionário são armazenar e recuperar valores a partir de chaves.

```
>>> tel = {"Alberto": 5432, "Fernanda": 5432, "Felisberto": 5321, "Godines":6453}
>>> tel["Godines"] #qual o telefone de Godines?
6453
>>> tel["Fernanda"] = 2000 #Fernanda mudou de telefone!
>>> tel
{'Godines': 6453, 'Fernanda': 2000, 'Felisberto': 5321, 'Alberto': 5432}
>>> tel.pop("Godines") #Godines não tem mais telefone
6453
>>> tel #ordenação em dicionário nunca é garantida
{'Fernanda': 2000, 'Felisberto': 5321, 'Alberto': 5432}
>>> print (tel.keys(),tel.values())#quais são as chaves e valores do dicionário?
(['Fernanda', 'Felisberto', 'Alberto'], [2000, 5321, 5432])
>>> tel.has_key("Felisberto")
True
>>> "Godines" in tel
False
```



python™

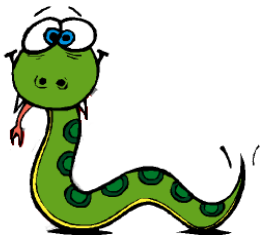


Estruturas de dados de alto nível

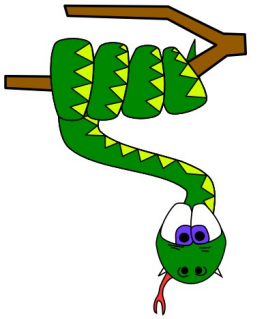
 # Dicionários (Mapeamentos / Tabelas Hash)

#Atualizando um dicionário:

```
>>> tel = {"Fernanda":2000, "Felisberto":5321, "Alberto":5432}
>>> mudancatel = {"Fernanda":2005, "Ana":3001, "Felisberto":3454}
>>> tel.update(mudancatel) #fazendo as atualizações de telefones
>>> tel
{'Fernanda': 2005, 'Ana': 3001, 'Alberto': 5432, 'Felisberto': 3454}
```



python™

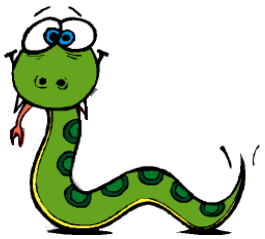


Funções

```
def nome_funcao (arg_1, arg_2, ..., arg_n):
        #
        # bloco de código contendo o corpo da função
        #
        return valor_de_retorno #retornar é opcional
```

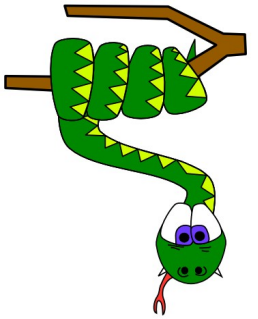
Ex.:

```
>>> def soimprimo(parametro):
...     print parametro
...
>>> soimprimo("RRRRRRRRRRRRRonaldiiiiiiiiinho!")
RRRRRRRRRRRRRonaldiiiiiiiiinho!
>>>
>>> soimprimo("Obina " + "vai voltar!")
Obina vai voltar!
```



python™

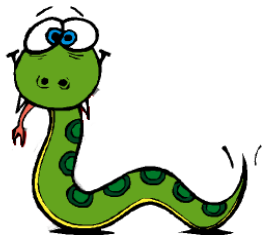
Funções



 # Valores padrão (default values)

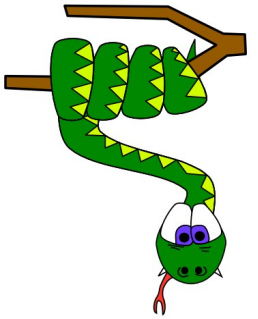
```
>>> def dividir(num=0.0, den=1.0):
...     print num/den
...
>>>
>>> dividir() #sem parâmetro, assume os valores padrão
0.0
>>> dividir(5,2) #num=5, den=2! Observe o resultado.
2
>>> dividir(den=4, num=16) #posso dizer explicitamente a quais parâmetros estou me referindo e com isso "mudar" a ordem dos parâmetros
4
>>> dividir(5) #num=5, den igual ao valor padrão (1.0)
5.0
```

Após valores padrão não podem haver outros parâmetros sem valor padrão.



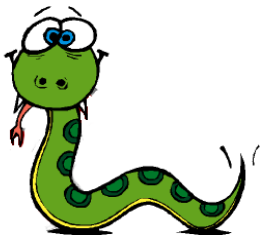
python™

Funções



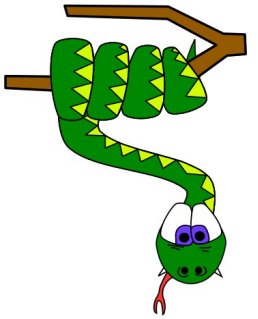
 # Múltiplos valores de retorno

```
>>> def retorna2(numero):
...     return numero,"Mensagem de status" #isso é uma forma de representar tuplas
...
>>> retorna2(500)
(500, 'Mensagem de status')
>>>
```



python™

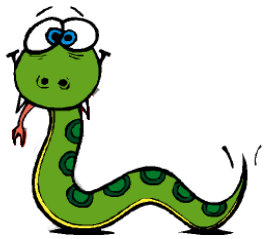
Funções



Parâmetros arbitrários

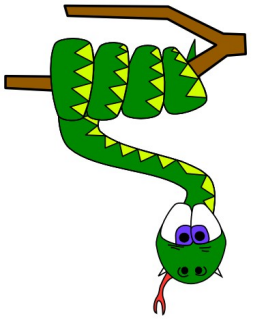
```
>>> def fexemplo(n, *opc, **opc_com_nome):
...     print "Tupla de argumentos opcionais:", opc
...     print "Dicionario de argumentos opcionais:", opc_com_nome
...
>>> fexemplo(1,2,3,4,a=95,c=43,texto="Textos sao textos, nada mais", d=(1,3))
Tupla de argumentos opcionais: (2, 3, 4)
Dicionario de argumentos opcionais: {'a': 95, 'c': 43, 'texto': 'Textos sao textos, nada mais', 'd': (1, 3)}
>>>
```

Valores opcionais 'sozinhos' entram na tupla de argumentos opcionais. Valores opcionais passados e atribuídos a uma variável são guardados no dicionário de argumentos opcionais.



python™

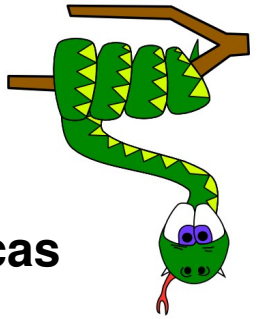
Funções e escopo de variáveis



```
#isso funciona
v = 1
def f():
        print v

#isso não funciona
v = 1
def f():
        print v
        v = 10
#[UnboundLocalError]

#isso faz o que você tentou fazer acima
v = 1
def f():
        global v
        print v
        v = 10
```

Variáveis globais só podem ser alteradas dentro de uma função se explicitamente ditas como globais.

python™

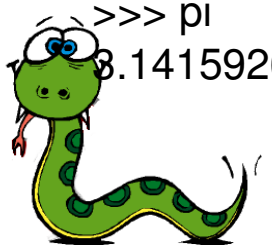


Módulos

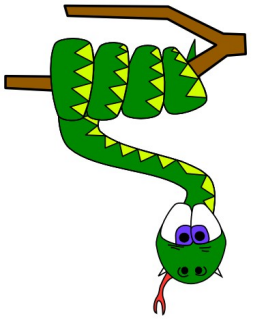
Módulo é um arquivo contendo código Python, são análogos às bibliotecas de C.

Importação de módulos

```
>>> import math #importa um módulo inteiro...
>>> math.sqrt(2) #... mas é necessário informar o nome do módulo para acessar seus componentes.
1.4142135623730951
>>>
>>> from math import sqrt #importa do módulo um componente específico
>>> sqrt(2) #os componentes estão acessíveis diretamente
1.4142135623730951
>>>
>>> from math import * #importando todo o módulo
>>> sqrt(2) #acesso direto a todos os componentes do módulo
1.4142135623730951
>>>
>>> from math import sqrt, pi #importando componentes específicos
>>> pi
3.1415926535897931
```



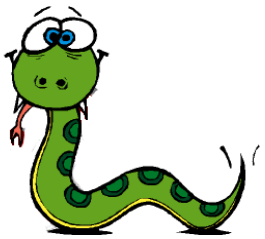
python™



Módulos

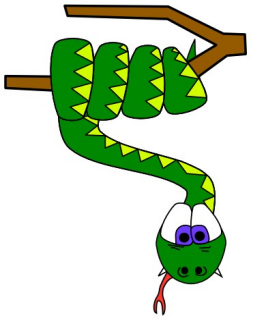
Listando os “componentes” de um módulo

>>> dir(math) **# na verdade, dir serve para listar os componentes de qualquer objeto (introspecção)**
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan', 'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp', 'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log', 'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh']

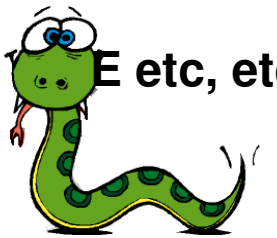


python™

Módulos padrão Python



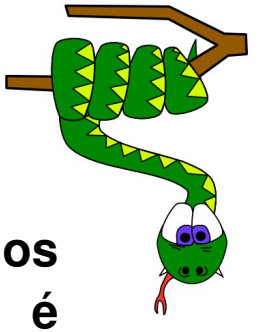
os – interface com o sistema operacional
email, smtplib – manipulação de emails, protocolo SMTP
pickle – persistência de objetos
re – expressões regulares
math – funções matemáticas reais
cmath – funções matemáticas complexas
random – geração de seleções aleatórias
pickle – persistência de dados
urllib2 – acesso a urls
zlib, gzip, bz2, zipfile, tarfile – compressão
gtk – biblioteca gráfica
datetime – manipulação de datas e intervalos de tempo
timeit – medição de desempenho de código
doctest, unitest – controle de qualidade, facilita o gerenciamento de testes
xml.dom, xml.sax, csv – suporte a xml, csv
threading – suporte a threads
gettext, locale, codecs – internacionalização (i18n)



E etc, etc, etc e mais etc.

python™

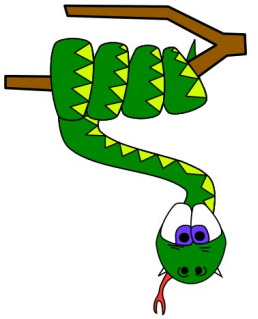
Trabalhando com arquivos (files)



Manipular arquivos é simples. Python provê o tipo 'file' e seus métodos facilitam o trabalho do programador. Em alguns aspectos, a sintaxe é parecida com a de C.

```
>>> f = open("meuarquivo.txt", "w") #abrindo arquivo com permissão de escrita
>>> f.write("Aqui estou,\nEscrevendo em arquivos!") #escrevendo...
>>> f.close() #fechando
>>> f = open("meuarquivo.txt","rw") #abrindo novamente em modo r/w
>>> f.read() #lê o arquivo inteiro
'Aqui estou,\nEscrevendo em arquivos!'
>>> f.seek(0) #reposiciona o 'apontador' de leitura para o início do arquivo
>>> f.readline() #lê uma linha de cada vez...
'Aqui estou,\n'
>>> f.readline() #outra...
'Escrevendo em arquivos!'
>>> f.readline() #outra tentativa! Retornou string vazia (fim de arquivo)
''
>>> f.close()
```

python™



Trabalhando com arquivos (files)

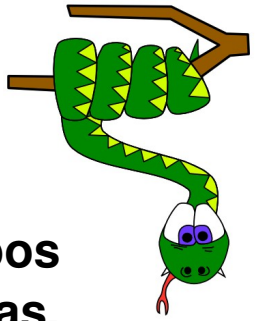
Principais métodos do tipo 'file'

f = open(“arq.txt”, “r”) **#abre para leitura**
f = open(“arq.txt”, “w”) **#abre para escrita**
f = open(“arq.txt”,”a”) **#abre para escrita ao final do arquivo**
f.close() **# fecha o arquivo**
texto = f.readlines() **#lê todas as linhas do arquivo e as copia para 'texto'**
linha = file.readline() **#copia a próxima linha do arquivo para a string 'linha'**
f.write(texto) **#escreve a string 'texto' no arquivo**
f.seek(posicao) **#posiciona o 'apontador' de r/w do arquivo para a posição (em bytes) especificada**

Outros métodos e mais informações: >>> help(file) **#divirta-se!**

Característica de leitura/escrita usando os métodos de 'file': Os métodos de leitura retornam strings, e os parâmetros dos métodos de escrita são strings. É necessário converter objetos de outros tipos (int, complex, tuple, etc.) para string antes de gravar em arquivos.

python™



Trabalhando com arquivos (Pickle)

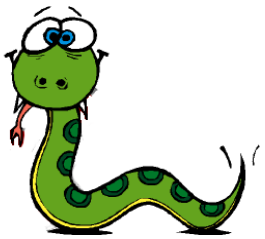
Strings podem ser facilmente escritas e lidas de um arquivo. Outros tipos precisam ser convertidos antes. Para estruturas de dados mais complexas, esse processo pode se tornar inconveniente e mais complicado.

Para que não sejam necessárias essas conversões, utiliza-se o módulo padrão pickle, que permite que praticamente qualquer objeto Python seja convertido para uma representação string, persistindo-o.

Representado como string, o objeto pode ser armazenado em arquivo, transferido pela rede, etc.

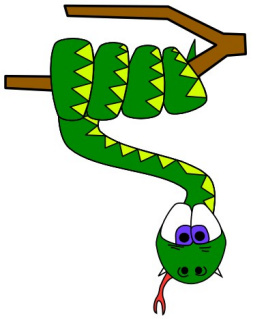
A forma de representação é humanamente entendível, embora não se recomende a manipulação direta da representação gerada com os métodos providos por 'file'.

Métodos principais de pickle: dump e load.



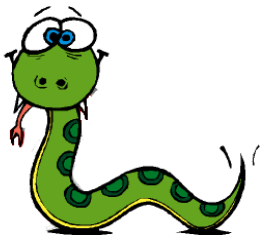
python™

Trabalhando com arquivos (Pickle)



Exemplo:

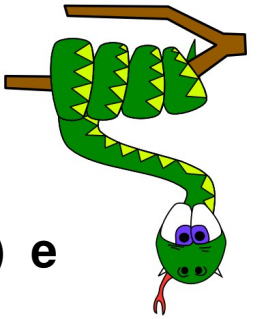
```
>>> f = open("exemplopickle.txt", "w")
>>> import pickle
>>> pickle.dump(54.3,f)
>>> pickle.dump([11,22,33,55],f)
>>> f.close()
>>>
>>> f = open("exemplopickle.txt", "rw")
>>> x = pickle.load(f)
>>> x
54.299999999999997
>>> type(x)
<type 'float'>
>>> y = pickle.load(f)
>>> y
[11, 22, 33, 55]
>>> type(y)
<type 'list'>
```



python™

Parte 2

Classes e objetos

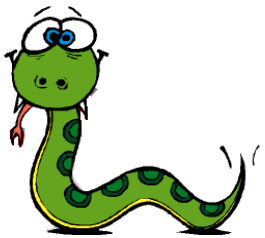


Classes agrupam um conjunto de objetos com propriedades (atributos) e "habilidade de ações" (métodos) semelhantes.

Objetos são instâncias das classes (cada objeto possui uma identidade).

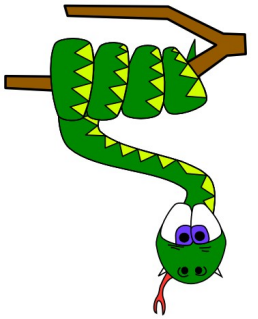
Sintaxe básica:

```
class nome_da_classe:
    #definição de funcionalidades e propriedades da classe
```



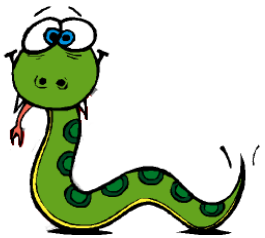
python™

Classes e objetos (exemplo 1)



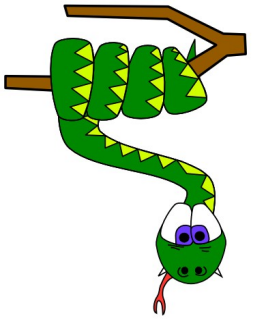
Exemplo 1:

```
#classe carro
class Carro:
        #atributos dos objetos da classe
        fabricante = “Ford”
        cor = “Vermelho”
        nome = “Fiesta”
        motor = 2.0
        estado = “ligado”

        #um método da classe
        def rebaixado():
                self.nome = “Premio”

        #um método que irei implementar depois (utilização de 'pass')
        def trocaMotor(motor):
                pass
```



python™

Classes e objetos (exemplo 1)

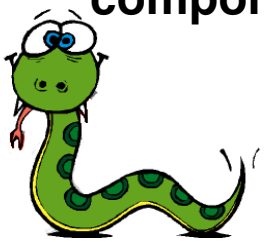


#instanciando um objeto
meupoize = Carro()

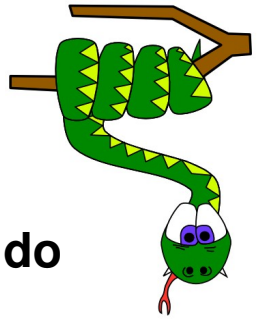
#acesso direto aos atributos da classe
meupoize.motor = “1.0”
meupoize.nome = “Fiat147”
meupoize.estado = “desligado”

Essa forma de se trabalhar com OO, permite que os atributos e métodos possam ser invocados diretamente por “agentes externos”. Nesse caso, os atributos, métodos e até a própria classe Carro do exemplo 1 são chamados de públicos.

Essa forma de acesso é bastante semelhante às structs de C, nas quais seus componentes são diretamente acessíveis.



python™



Métodos e atributos

Métodos definem as funcionalidades da classe. São análogos às funções do "mundo procedural".

Atributos definem as propriedades da classe.

Em Python, um método ou atributo pode ser público ou privado.

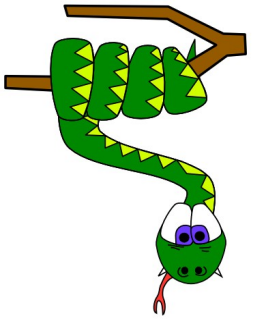
Os "componentes" públicos são acessíveis diretamente por qualquer outro objeto.

Os "componentes" privados só podem ser acessado internamente por componentes do próprio objeto, não sendo acessível diretamente a nenhum outro objeto.

Diferentemente de Java, não há definição da "permissão" protegido (protected).

A sintaxe para informar que um dado componente é privado é iniciar seu nome com dois sublinhados.

python™



Métodos e atributos

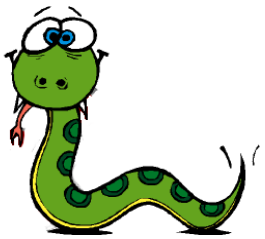
#declarando “componentes” como privados

__atributoprivado = 1
__metodoprivado()

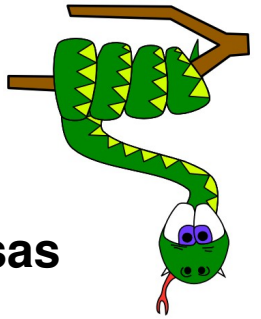
No exemplo 2, serão vistos alguns detalhes sobre os componentes privados em Python. Será aproveitado também para mostrar a utilização do método __init__() [que é análogo ao construtor de outras linguagens].

Também será observada a utilização da palavra “self”, que tem como função explicitar que estamos nos referindo a componentes do próprio objeto (é análogo ao 'this' de Java).

Ah!, o exemplo 2 (classeCarroex2.py) está disponível apenas em arquivo-fonte, não sendo colocado diretamente nessa apresentação.



python™

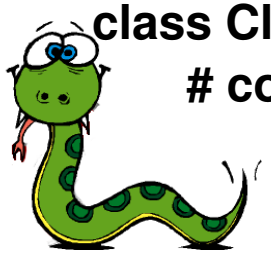


Herança e herança múltipla

Herança é um mecanismo que permite que características comuns a diversas classes sejam fatoradas em uma classe base, ou superclasse.
Cada classe derivada (subclasse) apresenta as características da superclasse e acrescenta ou sobrescreve (mecanismo de sobrecarga) o que for definido de particularidades para ela.

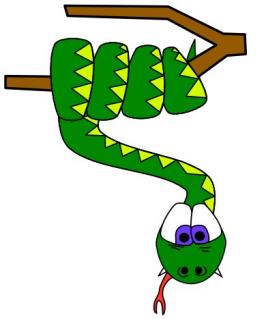
Sintaxe:

```
class ClasseBase1:
    # corpo da classe base 1

class ClasseBase2:
    # corpo da classe base 2

class ClasseDerivada1(ClasseBase1):
    # corpo da classe derivada 1 [exemplo de herança simples]

class ClasseDerivada2(ClasseBase1, ClasseBase2):
    # corpo da classe derivada 2 [exemplo de herança múltipla]
```



python™

Observações importantes



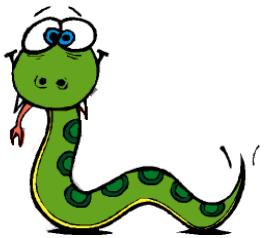
Para verificar se um objeto pertence a uma determinada classe ou tipo:

isinstance(objeto, classe/tipo)

```
>>> isinstance([1,2,3],list)
True
>>> isinstance(1.4,int)
False
>>>
```

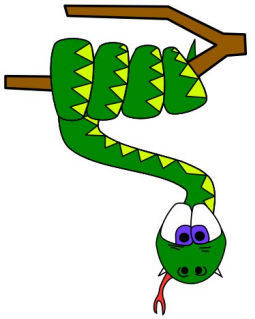
Para verificar se uma classe é subclasse de outra:

issubclass(subclasse, superclasse) #retorna verdadeiro também se o primeiro e o segundo argumento forem a própria classe.



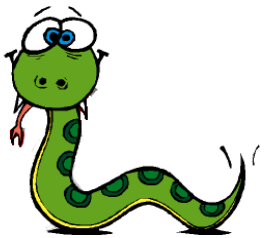
python™

Observações importantes



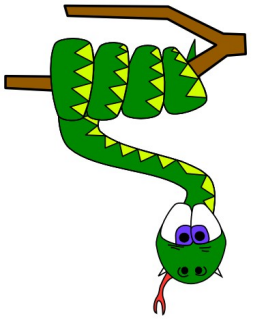
Para copiar objetos:

```
>>> import copy
>>> copia = copy.copy(objeto_a_ser_copiado)
```



python™

Propriedades, classe base e herança



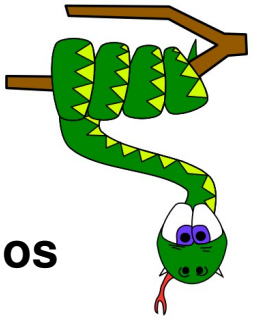
Há uma maneira diferente de fazer os métodos get e set.

Exemplo:

```
class ClasseQualquer(object):
        def __init__(self, parametro = None):
                self.__qualquer = parametro

        def getQualquer(self):
                return self.__qualquer

        def setQualquer(self, argumento):
                self.__qualquer = argumento

        privado = property(getQualquer, setQualquer)

objeto = ClasseQualquer()
print objeto.privado
objeto.privado = -32
print objeto.privado
```

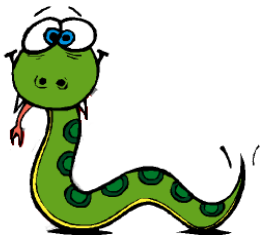
python™

Atributos e métodos estáticos



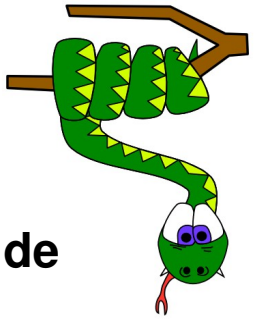
Métodos e atributos são ditos estáticos quando são comuns a todos os objetos de uma classe.

```
class ClasseFulana(object):
	contador = 0

	def __init__(self):
		ClasseFulana.contador += 1 #atributo estático
		print "Classe instanciada pela",ClasseFulana.contador,"a. vez\n"

	def qtdInstancias():
		print "A classe ClasseFulana  possui", ClasseFulana.contador,
"instancia(s)"

	instancias = staticmethod(qtdInstancias) #declarando um método como
estático
```



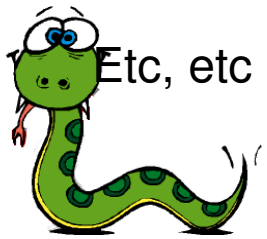
python™

Métodos especiais e sobrecarga



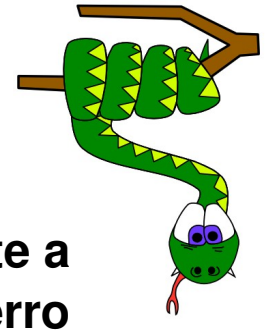
Em Python, a sobrecarga de operadores é feita a partir da definição de métodos especiais da classe.

__add__ -> sobrecarga da adição
__sub__ -> sobrecarga da subtração
__mul__ -> sobrecarga da multiplicação
__div__ -> sobrecarga da divisão
__mod__ -> sobrecarga do resto
__pow__ -> sobrecarga da potência (**)
__iadd__ -> sobrecarga de +=
__isub__ -> sobrecarga de -=
__imul__ -> sobrecarga de *=
__idiv__ -> sobrecarga de /=
__lt__ -> sobrecarga de <
__eq__ -> sobrecarga de ==
__cmp__ -> sobrecarga do comparador (ordenação)
__str__ -> sobrecarga da “formatação” do print



Etc, etc e etc...

python™



Mecanismo de exceções

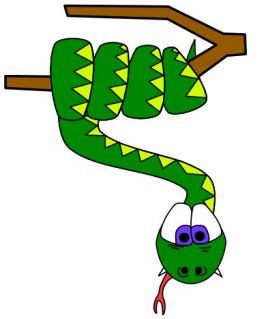
Exceções indicam que algum tipo de condição excepcional ocorreu durante a execução do programa, logo, exceções estão associadas a condições de erro em tempo de execução.

Elas podem ser geradas/lançadas (quando é sinalizado que uma condição excepcional ocorreu) e capturadas (quando é feita a manipulação (tratamento) da situação excepcional, onde as ações necessárias para a recuperação de situação de erro são definidas.

Na prática, acontece que a maioria das exceções não é tratada e acabam resultando em mensagens de erro. Exemplos:

```
>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> f = open("arquivoquenaoexiste","r")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IOError: [Errno 2] No such file or directory: 'arquivoquenaoexiste'
```

python™

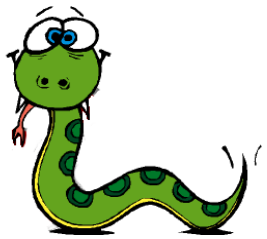


Mecanismo de exceções

Exemplos:

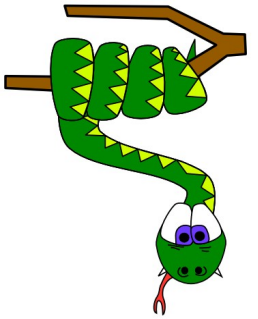
```
>>> "string" + 3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
>>>
>>> a = int(raw_input("Entre com um numero: "))
Entre com um numero: teimoso
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: invalid literal for int(): teimoso
>>>
```

Etc, etc e etc...



python™

Tratamento de exceções

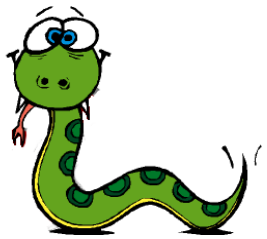


```
while True:
    try:
        x = int(raw_input("Nobre senhor e senhora, digite um numero: "))
        break
    except ValeuError:
        print "Meu senhor... minha senhora... Isso não é um número válido. Vai lá, entre outra vez!"
```

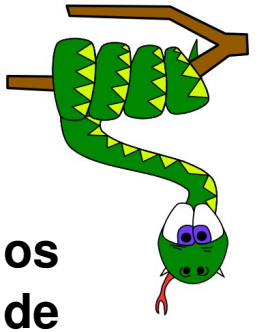
O try funciona da seguinte forma:

Primeiramente, o conjunto de comandos entre as palavras reservadas try e except (denominada de cláusula try) é executado.

Se não for gerada exceção, a cláusula except é ignorada e termina a execução da construção try.



python™



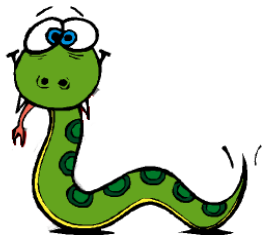
Tratamento de exceções

Se ocorrer uma exceção durante a execução da cláusula try, os comandos que ainda faltam ser executados nela são ignorados. Se o tipo de exceção ocorrida tiver sido previsto com alguma cláusula except, então essa cláusula será executada. Ao fim da cláusula também termina a execução do try como um todo.

Se a exceção ocorrida não foi prevista em nenhum tratador except da construção try em que ocorreu, então ela é entregue a uma construção try mais externa. Se não existir nenhum tratador previsto para tal exceção (exceções não-manipuladas), a execução encerra com uma mensagem de erro.

Uma observação oportuna é que pode-se tratar várias exceções diferentes de mesmo modo utilizando-se uma única cláusula except.

Exceções podem ser lançadas com a palavra reservada 'raise'.



python™