

Luciano Ramalho
ramalho@python.pro.br

@pythonprobr



outubro/2013

Objetos Pythonicos

Orientação a objetos e padrões de projeto em Python

Para me encontrar

- Juntem-se ao grupo no Google Groups
 - solicite um convite para ramalho@python.pro.br
- Estou à disposição de vocês pelo e-mail acima e:
 - cel: 11-9-8432-0333 (não deixe recado, use SMS)
 - skype: LucianoRamalho (combinar via e-mail)
 - GTalk: ramalho@python.pro.br (combinar via e-mail)

Para me encontrar (2)

- Eventos:
 - PythonBrasil [8], FISL, TDC etc.
- Listas de discussão da comunidade:
 - python-brasil, django-brasil, grupy-sp
- Garoa Hacker Clube (1º hackerspace do Brasil)
 - <http://garoa.net.br>
 - grupo: hackerspacesp

Funcionamento do curso

- seis aulas online ao vivo de 2h cada
 - os vídeos poderão ser baixados no dia seguinte (há um processamento a fazer, por isso demora)
- realizar tarefas antes da próxima aula, quando houver
- discutir dúvidas, colocar questões mais avançadas e ajudar colegas no grupo

Temas I:

Fundamentos

- Introdução: explorando objetos nativos
- Terminologia básica de OO no jargão do Python
- Regras de escopo; atributos de classes x instâncias
- Polimorfismo e duck typing: conceitos e exemplos

Temas 2:

Objetos Pythonicos

- Encapsulamento: filosofia, getters/setters, propriedades
- Herança múltipla: conceito e exemplo simples
- Exemplos de APIs com herança múltipla
 - Tkinter GUI e Django generic views

Temas 3:

Python Object Model

- Sobrecarga de operadores: conceito e exemplos simples
- Sequências, iteráveis e geradores
- Exemplos de uso de sobrecarga de operadores: Vetor, Bunch, Django ORM e Tkinter GUI

Temas 3:

Padrões de Projeto

- Funções como objetos e objetos invocáveis
- Protocolos (interfaces informais)
- Classes abstratas
- Alguns padrões de projeto e sua aplicação em Python

Temas 4:

Metaprogramação

- Decoradores de métodos e de classes
- Descritores de atributos
- Exemplo de API com descritores: Django models

Aula I

Introdução à Orientação Objetos em Python

Objetivos desta aula

- Apresentar conceitos fundamentais de orientação a objetos, utilizando a terminologia da comunidade Python
- Explicar conceitos a partir do nível básico para:
 - programadores com pouca experiência em OO
 - programadores que já usaram OO em outras linguagens

Orientação a objetos: a origem

- Linguagem Simula 1967
 - Noruega: Ole-Johan Dahl e Kristen Nygaard
 - objetos, classes, sub-classes
 - métodos virtuais (funções associadas a objetos específicos em tempo de execução)

Orientação a objetos: evolução

- Smalltalk 1980
 - EUA, Xerox PARC (Palo Alto Research Center): Alan Kay, Dan Ingalls, Adele Goldberg et. al.
- terminologia:
 - “Object oriented programming”
 - “message passing”, “late binding” (a idéia por trás de métodos virtuais)

System Browser

Collections-Sequences

Collections-Text

Collections-Arrays

Collections-Streams

Collections-Support

Graphics-Primitives

Graphics-Display

Graphics-Media

Graphics-Paths

Interval

LinkedList

MappedCollection

OrderedCollection

SortedCollection

accessing

copying

adding

removing

enumerating

private

collect:

do:

do:andBetweenDo:

promoteFirstSuchT

reverse

reverseDo:

select:

Form Editor

instance

class

collect: aBlock

"Evaluate aBlock with each of my elements as the argument. Collect the resulting values into a collection that is like me. Answer with the collection. Override superclass in order to use add:, not at:put:."

| newCollection |

newCollection ← self species new.

self do: [:each | newCollection add: (aBlock value: each)].

newCollection

User Interrupt

Paragraph>>characterBlockAtPoint:

Paragraph>>mouseSelectto:

CodeController(ParagraphEditor)>>processRedButton

CodeController(ParagraphEditor)>>processMouseButtons

CodeController(ParagraphEditor)>>controlActivity

CodeController(Controller)>>controlLoop

controlActivity

self scrollbarContainsCursor

ifTrue:

[self scroll]

ifFalse:

[self processKeyboard]

self processMouseButtons

blueButton

scrollBar

marker

savedArea

paragraph

startBlock

31@537 corner:

63@770

File List

[<Robson>SF>*]

[File]<Robson>SF>ScreenForm.st

[File]<Robson>SF>ScreenForm.text

[File]<Robson>SF>ScreenFormChanges.st

[File]<Robson>SF>WordGraphics.form

Rectangle fromUser origin

ScreenForm setFullPageWidth.

ScreenForm

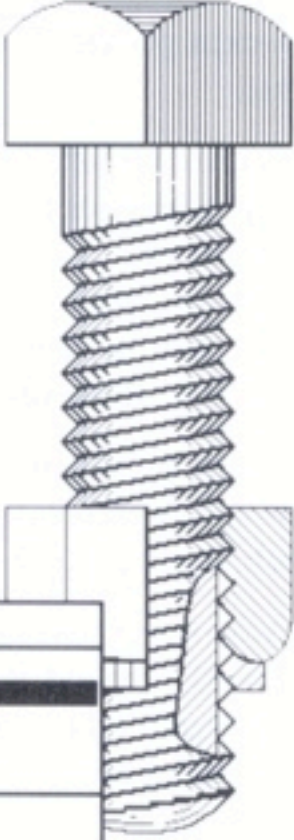
printRectangle:

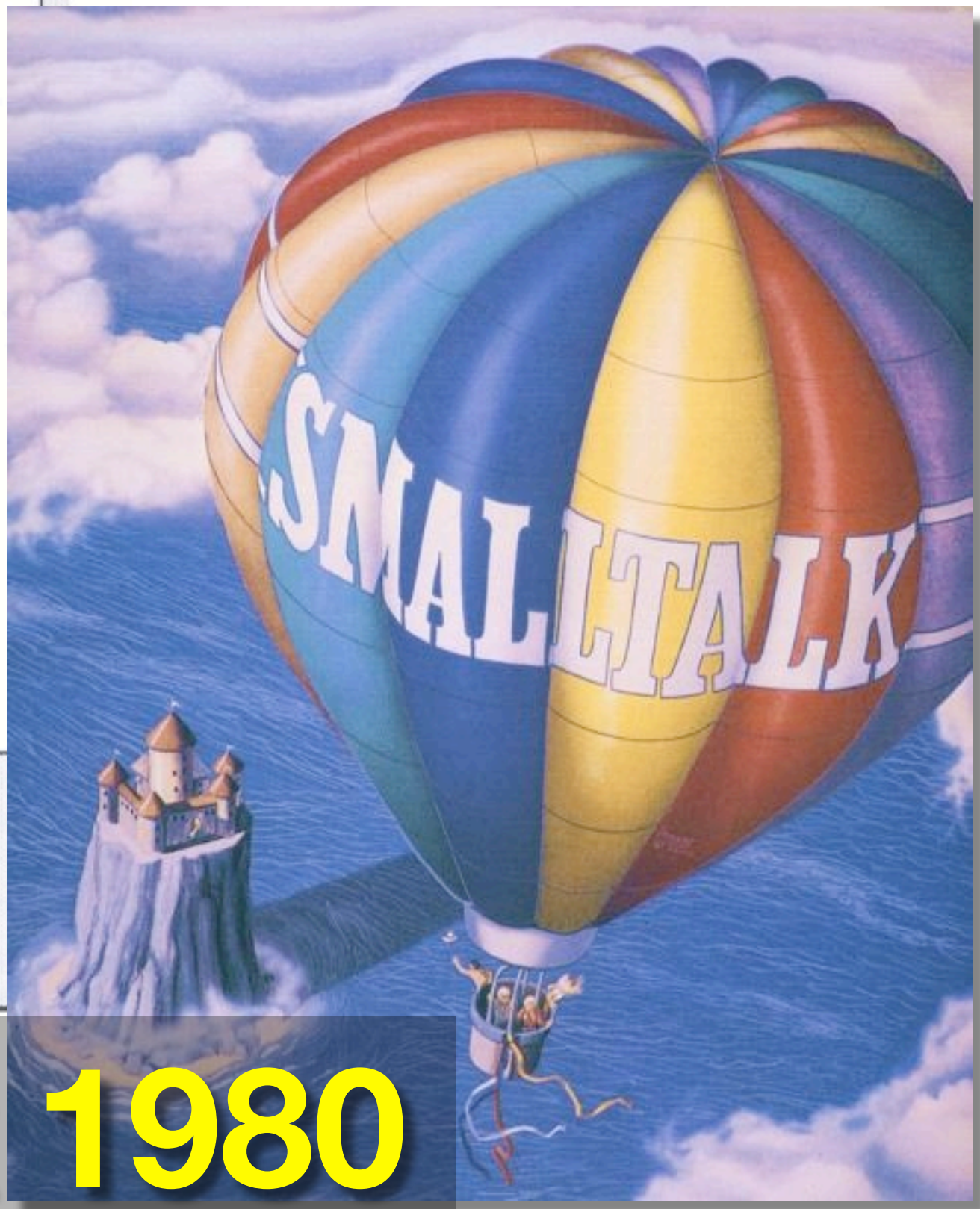
(30@5 extent: 674@790)

onFileNamed: 'ExampleScreen.press'

(Form readFrom: 'FilledSkate.form') edit

Fig.1.





Smalltalk 1980

SqueakMap Package Loader (544/544) O

README-SoapCore Client O

Workspace O

```
TimeProfileBrowser spyOn:
[Transcript show: (100
factorial) printString]
```

Transcript O

```
14847794933262154439441526816992388
56266700490715968264381621468592963
89521759999322991560894146397615651
82862536979208272237582511852109168
64000000000000000000000000000000
```

Hierarchy Browser: Number O

Kernel-Numbers

```
ProtoObject
Object
Magnitude
Number
Float
Fraction
Integer
LargePositiveInteger
LargeNegativeInteger
SmallInteger
ScaledDecimal
```

```
-- all --
arithmetic
mathematical functions
truncation and round off
testing
converting
intervals
printing
comparing
filter streaming
vocabulary
```

```
abs
adaptToCollection:andSend:
adaptToFloat:andSend:
adaptToFraction:andSend:
adaptToInteger:andSend:
adaptToPoint:andSend:
adaptToScaledDecimal:andSend:
adaptToString:andSend:
arcCos
arcSin
arcTan
arcTan:
arg
asB3DVector3
asDuration
asFloatD
asFloatE
```

instance ? class

browse senders implementors versions inheritance hierarchy inst vars class vars source

```
Magnitude subclass: #Number
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'Kernel-Numbers'
```

Class Number holds the most general methods for dealing with numbers. Subclasses Float, Fraction, and Integer, and their subclasses, provide concrete representations of a numeric quantity.

All of Number's subclasses participate in a simple type coercion mechanism that supports mixed mode arithmetic and comparisons. It works as follows. If a comparison or arithmetic operation fails because of incompatible types, it is retried in the following guise:

```
self asFloatD <= 4:3
```

This gives the arg or the B an opportunity to resolve the incompatibility, knowing exactly what two types are involved. If self is more general, then arg will be converted and

Time Profile O

- 143 tallies, 186 msec.

Tree

```
97.9% (182ms) TranscriptStream>>show:
97.9% (182ms) TranscriptStream>>endEntry
97.9% (182ms) TranscriptStream(Object)>>changed:
97.9% (182ms) PluggableTextMorph>>update:
97.9% (182ms) PluggableTextMorph(Morph)>>refreshWorld
97.9% (182ms) PasteUpMorph>>displayWorldSafely
97.9% (182ms) WorldState>>displayWorldSafely:
97.9% (182ms) PasteUpMorph>>displayWorld
97.9% (182ms) PasteUpMorph>>privateOuterDisplayWorld
97.9% (182ms) WorldState>>displayWorld:submorphs:
97.9% (182ms) WorldState>>drawWorld:submorphs:invalidAreasOn:
97.9% (182ms) FormCanvas(Canvas)>>fullDrawMorph:
97.9% (182ms) FormCanvas(Canvas)>>fullDraw:
97.9% (182ms) SystemWindow(Morph)>>fullDrawOn:
81.8% (152ms) SystemWindow(Morph)>>drawSubmorphsOn:
81.8% (152ms) FormCanvas(Canvas)>>fullDrawMorph:
81.8% (152ms) FormCanvas(Canvas)>>fullDraw:
81.8% (152ms) PluggableMessageCategoryListMorph(Morph)>>fullDrawOn:
53.1% (99ms) PluggableMessageCategoryListMorph(Morph)>>drawSubmorphsOn:
81.8% (152ms) FormCanvas(Canvas)>>fullDrawMorph:
```

browse senders implementors versions inheritance hierarchy inst vars class vars source

endEntry

"Display all the characters since the last endEntry, and reset the stream"

```
self semaphore critical:[
self changed: #appendEntry.
self reset.
].
```

System Browser: SoapHttpClient O

```
SOAP-DataBinding
SOAP-Encoding
SOAP-Decoding
SOAP-ClientBase
SOAP-Http
SOAP-Http-Client
SOAP-RPC
SOAP-Example
SOAP-GoogleWebAPI
SOAP-CustomComplexTy
```

```
SoapContentTypeMismatch
SoapHttpClient
SoapHttpConnector
SoapHttpGenericError
```

```
-- all --
initialize-release
actions
private
accessing
constants
actions-hooks
```

```
defaultPort
getResponseContents
host:port:page:
parseURL:
prepareAdditional:
prepareEnvelope:
prepareSocket
proxyPort
proxyServer
redirectWith:
sendBySocket
send:action:
```

instance ? class

browse senders implementors versions inheritance hierarchy inst vars class vars source

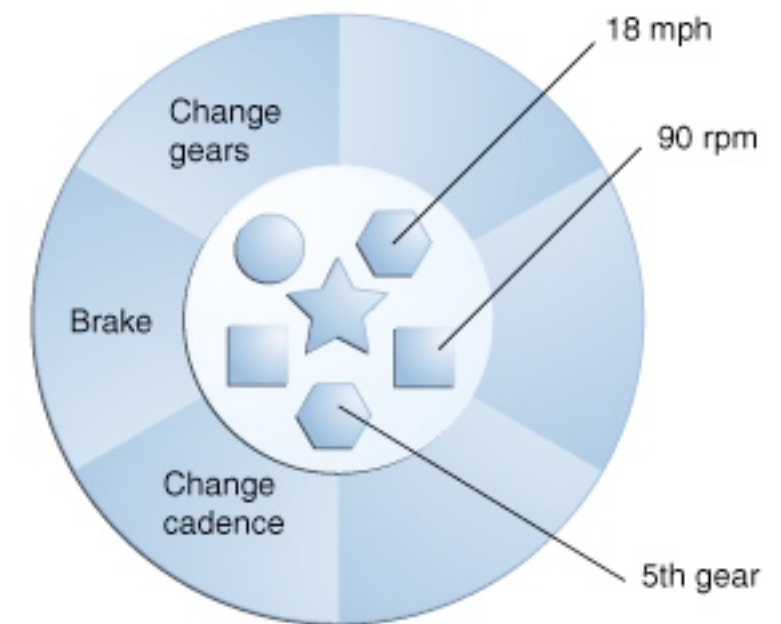
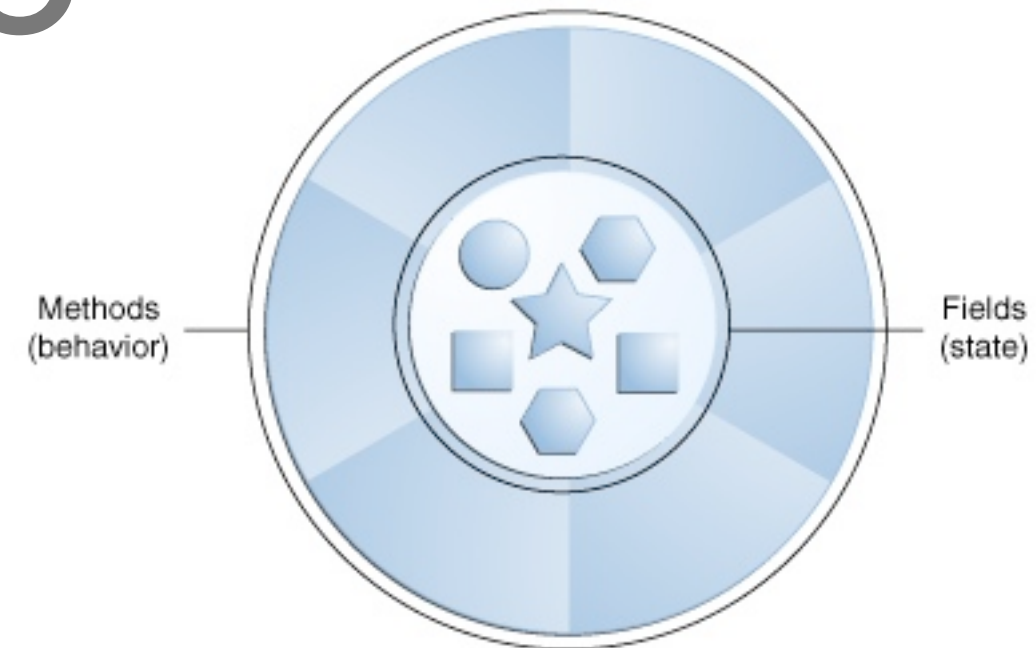
prepareSocket

```
socket := HTTPSocket new.
socket connectTo: serverAddress port: port.
socket
waitForConnectionUntil: (Socket deadlineSecs: self class connectionTimeout).
socket isConnected
ifFalse: [self cannotConnectToServer signal: 'Server did not respond - 'host: ', host, '
port: ', port printString]
```

Squeak: Smalltalk livre

Conceito: “objeto”

- Um componente de software que inclui dados (atributos) e comportamentos (métodos)
- Em geral, os atributos são manipulados pelos métodos do próprio objeto (encapsulamento)



Figuras: bycycle (bicicleta), The Java Tutorial

<http://docs.oracle.com/javase/tutorial/java/concepts/object.html>

Terminologia Pythonica

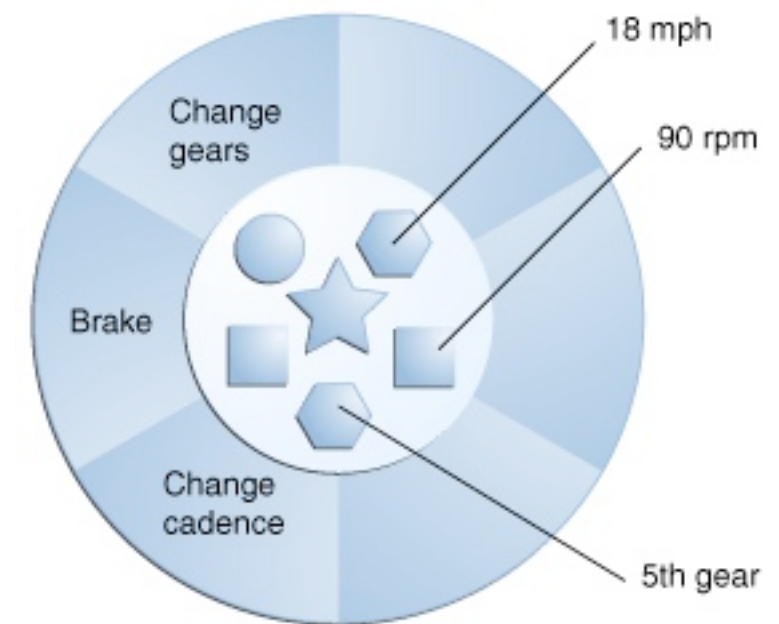
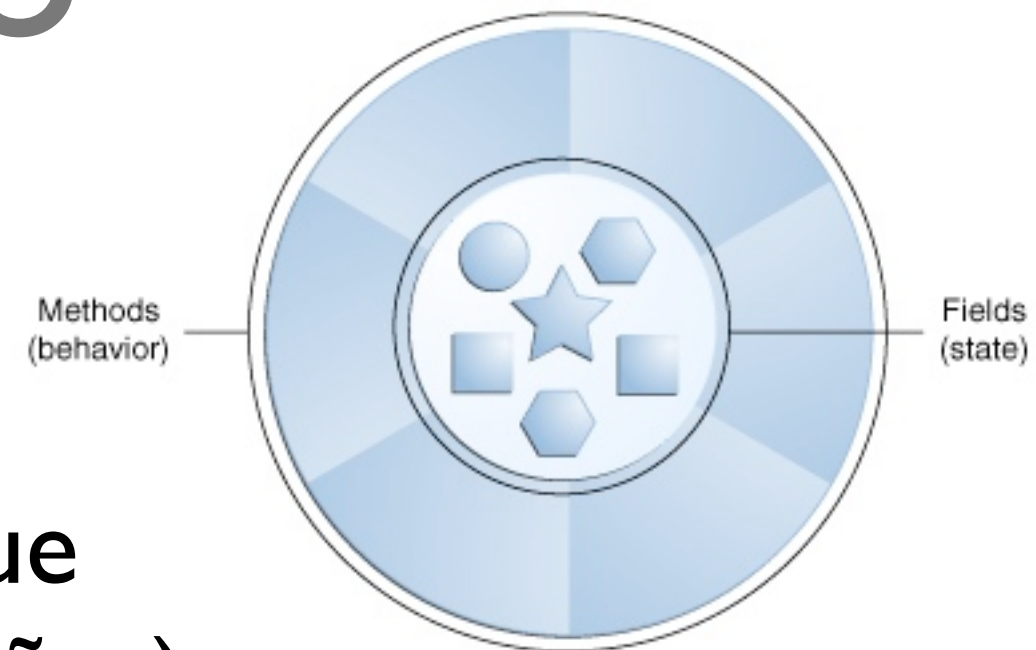
- Python tem um modelo unificado: todas as funções são objetos, assim como números, strings etc. são objetos
- Isso significa que os métodos também são atributos (especificamente: das classes)
- assim como os “campos” numéricos, string etc. dos objetos são atributos

Terminologia Pythonica (2)

- Então “campos” e “métodos” são chamados igualmente de atributos
- Quando se quer falar apenas de “campos” usa-se o termo atributo de dados (data attribute)

Conceito: “objeto”

- Um componente de software que inclui **atributos de dados** e **métodos** (que nada mais são que atributos invocáveis como funções)
- Em geral, os **atributos de dados** são manipulados pelos **métodos** do próprio objeto (encapsulamento)



Figuras: bycycle (bicicleta), The Java Tutorial

<http://docs.oracle.com/javase/tutorial/java/concepts/object.html>

Exemplo: um objeto dict

```
>>> d = {'AM': 'Manaus', 'PE': 'Recife', 'PR': 'Curitiba'}
>>> d.keys()
['PR', 'AM', 'PE']
>>> d.get('PE')
'Recife'
>>> d.pop('PR')
'Curitiba'
>>> d
{'AM': 'Manaus', 'PE': 'Recife'}
>>> len(d)
2
>>> d.__len__()
2
```

- Métodos: keys, get, pop, __len__ etc.

Exemplo: um objeto dict

- Sobrecarga de operadores:
- `[]: __getitem__, __setitem__`

```
>>> d
{'AM': 'Manaus', 'PE': 'Recife'}
>>> d['AM']
'Manaus'
>>> d.__getitem__('AM')
'Manaus'
>>> d['MG'] = 'Belo Horizonte'
>>> d.__setitem__('RJ', 'Rio de Janeiro')
>>> d
{'MG': 'Belo Horizonte', 'AM': 'Manaus', 'RJ': 'Rio de Janeiro', 'PE': 'Recife'}
```

Exemplo: um objeto dict

- Atributos de dados: `__class__`, `__doc__`

```
>>> d.__class__
<type 'dict'>
>>> print d.__doc__
dict() -> new empty dictionary.
dict(mapping) -> new dictionary initialized from a mapping object's
    (key, value) pairs.
dict(seq) -> new dictionary initialized as if via:
    d = {}
    for k, v in seq:
        d[k] = v
dict(**kwargs) -> new dictionary initialized with the name=value pairs
    in the keyword argument list.  For example:  dict(one=1, two=2)
```

Exemplo: um objeto dict

- Em Python, métodos também são atributos

```
>>> dir(d)
['__class__', '__cmp__', '__contains__', '__delattr__', '__delitem__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__',
 '__hash__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__ne__',
 '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
 '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'clear', 'copy',
 'fromkeys', 'get', 'has_key', 'items', 'iteritems', 'iterkeys', 'itervalues',
 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']
```

Exemplo: um objeto Tkinter.Label

```
import Tkinter
from time import strftime

relogio = Tkinter.Label()

relogio.pack()
relogio['font'] = 'Helvetica 120 bold'
relogio['text'] = strftime('%H:%M:%S')

def tictac():
    agora = strftime('%H:%M:%S')
    if agora != relogio['text']:
        relogio['text'] = agora
    relogio.after(100, tictac)

tictac()
relogio.mainloop()
```



Note:

Em Tkinter, atributos de dados são acessados via `[]`: `__getitem__` e `__setitem__`

Objetos em linguagens

- Existem linguagens “baseadas em objetos” e linguagens “orientadas a objetos”
- baseadas em objetos: permitem que o programador use os tipos de objetos fornecidos, mas não permitem que ele crie seus próprios tipos de objetos
- Ex. Visual Basic antes da era .net

Objetos em Python

- Tudo é objeto: não existem “tipos primitivos”
 - desde Python 2.2, dezembro de 2001

```
>>> 5 + 3
8
>>> 5 . __add__(3)
8
>>> type(5)
<type 'int'>
```

Funções são objetos

```
>>> def fatorial(n):  
...     '''devolve n!'''  
...     return 1 if n < 2 else n * fatorial(n-1)  
...  
>>> fatorial(5)  
120  
>>> fat = fatorial  
>>> fat  
<function fatorial at 0x1004b5f50>  
>>> fat(42)  
1405006117752879898543142606244511569936384000000000L  
>>> fatorial.__doc__  
'devolve n!'  
>>> fatorial.__name__  
'fatorial'  
>>> fatorial.__code__  
<code object fatorial at 0x1004b84e0, file "<stdin>", line 1>  
>>> fatorial.__code__.co_varnames  
('n',)
```

Funções são objetos

```
>>> fatorial.__code__.co_code
'|\\x00\\x00d\\x01\\x00j\\x00\\x00o\\x05\\x00\\x01d\\x02\\x00S\\x01|\\x00\\x00t\\x00\\x00|\\x00\\x00d\\x02\\x00\\x18\\x83\\x01\\x00\\x14S'
>>> from dis import dis
>>> dis(fatorial.__code__.co_code)
      0 LOAD_FAST           0 (0)
      3 LOAD_CONST          1 (1)
      6 COMPARE_OP           0 (<)
      9 JUMP_IF_FALSE        5 (to 17)
     12 POP_TOP
     13 LOAD_CONST           2 (2)
     16 RETURN_VALUE
>>  17 POP_TOP
     18 LOAD_FAST           0 (0)
     21 LOAD_GLOBAL          0 (0)
     24 LOAD_FAST           0 (0)
     27 LOAD_CONST           2 (2)
     30 BINARY_SUBTRACT
     31 CALL_FUNCTION         1
     34 BINARY_MULTIPLY
     35 RETURN_VALUE
>>>
```

Bytecode da
função fatorial

Objetos têm tipo

- Tipagem forte: normalmente, Python não faz conversão automática entre tipos

```
>>> a = 10
>>> b = '9'
>>> a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> a + int(b)
19
>>> str(a) + b
'109'
>>> 77 * None
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for *: 'int' and 'NoneType'
>>>
```

Tipagem dinâmica: variáveis não têm tipo

```
>>> def dobro(x):  
...     return x * 2  
...  
>>> dobro(7)  
14  
>>> dobro(7.1)  
14.2  
>>> dobro('bom')  
'bombom'  
>>> dobro([10, 20, 30])  
[10, 20, 30, 10, 20, 30]  
>>> dobro(None)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
  File "<stdin>", line 2, in dobro  
TypeError: unsupported operand type(s) for *: 'NoneType' and 'int'
```

Duck typing

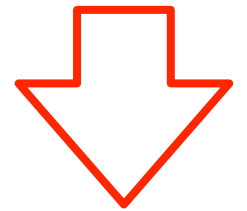
- “Se voa como um pato, nada como um pato e grasna como um pato, é um pato.”
- Tipagem dinâmica permite duck typing (tipagem pato) estilo de programação que evita verificar os tipos dos objetos, mas apenas seus métodos
- No exemplo anterior, a função dobro funciona com qualquer objeto x que consiga fazer $x * 2$
- x implementa o método `__mult__(n)`, para n inteiro

Tipagem forte x fraca

- Tipagem forte x fraca refere-se a conversão automática de valores de tipos diferentes.
- Linguagens de tipagem fraca são muito liberais na mistura entre tipos, e isso é uma fonte de bugs.

Veja alguns resultados estranhos obtidos com JavaScript, que tem tipagem fraca. Em Python as três expressões acima geram `TypeError`, e as três últimas resultam `False`. Python tem tipagem forte.

JavaScript
(ECMAScript 5)
em Node.js 0.6



```
> 10 + '9'
'109'
> 10 + '9' * 1
19
> '10' + 9 * 1
'109'
```

```
> 0 == '0'
true
> 0 == ''
true
> '0' == ''
false
```


Tipagem forte x fraca, dinâmica x estática

- Tipagem forte x fraca refere-se a conversão automática de valores de tipos diferentes
- Tipagem dinâmica x estática refere-se à declaração dos tipos das variáveis, parâmetros formais e valores devolvidos pelas funções
- Linguagens de tipagem estática exigem a declaração dos tipos, ou usam inferência de tipos para garantir que uma variável será associada a somente a valores de um tipo

Tipagem em linguagens

Smalltalk	dinâmica	forte
Python	dinâmica	forte
Ruby	dinâmica	forte
C (K&R)	estática	fraca
C (ANSI)	estática	forte
Java	estática	forte
C#	estática	forte
JavaScript	dinâmica	fraca
PHP	dinâmica	fraca

} combinação
perigosa:
bugs sorrateiros

Conversões automáticas

- Python faz algumas (poucas) conversões automáticas entre tipos:
- Promoção de int para float
- Promoção de str para unicode
 - assume o encoding padrão: ASCII por default

```
>>> 6 * 7.0
42.0
>>> 'Spam, ' + u'eggs'
u'Spam, eggs'
>>>
```

Objetos podem receber novos atributos

- Em geral, é possível atribuir valores a atributos não pré-definidos, em tempo de execução.
- Exceções: tipos embutidos, classes com `__slots__`

```
>>> fatorial
<function fatorial at 0x1004b5f50>
>>> fatorial._autor = 'Fulano de Tal'
>>> fatorial._autor
'Fulano de Tal'
>>> s = 'sapo'
>>> s.nome = 'Joca'
Traceback (most recent call last):
...
AttributeError: 'str' object has no attribute 'nome'
```

Conceito: “classe”

- Uma categoria, ou tipo, de objeto
 - Uma idéia abstrata, uma forma platônica
- Exemplo: classe “Cão”:
 - Eu digo: “Ontem eu comprei um cão”
 - Você não sabe exatamente qual cão, mas sabe:
 - é um mamífero, quadrúpede, carnívoro
 - pode ser domesticado (normalmente)
 - cabe em um automóvel

Exemplar de cão: instância da classe Cao



```
>>> rex = Cao()
```

instanciação

@pythonprobr

instanciação

Classe Cao

```
class Mamifero(object):  
    """lição de casa: implementar"""  
  
class Cao(Mamifero):  
    qt_patas = 4  
    carnivoro = True  
    nervoso = False  
    def __init__(self, nome):  
        self.nome = nome  
    def latir(self, vezes=1):  
        # quando nervoso, late o dobro  
        vezes = vezes + (self.nervoso * vezes)  
        print self.nome + ':' + ' Au!' * vezes  
    def __str__(self):  
        return self.nome  
    def __repr__(self):  
        return 'Cao({0!r})'.format(self.nome)
```

```
>>> rex = Cao('Rex')  
>>> rex  
Cao('Rex')  
>>> print rex  
Rex  
>>> rex.qt_patas  
4  
>>> rex.latir()  
Rex: Au!  
>>> rex.latir(2)  
Rex: Au! Au!  
>>> rex.nervoso = True  
>>> rex.latir(3)  
Rex: Au! Au! Au! Au! Au! Au!
```

Classe Cao em Python

```
class Mamifero(object):  
    """lição de casa: implementar"""  
  
class Cao(Mamifero):  
    qt_patas = 4  
    carnivoros = True  
    nervoso = False  
    def __init__(self, nome):  
        self.nome = nome  
    def latir(self, vezes=1):  
        # quando nervoso, late o dobro  
        vezes = vezes + (self.nervoso * vezes)  
        print self.nome + ':' + ' Au!' * vezes  
    def __str__(self):  
        return self.nome  
    def __repr__(self):  
        return 'Cao({0!r})'.format(self.nome)
```



atributos de dados na classe funcionam como valores default para os atributos das instâncias

`__init__` é o construtor, ou melhor, o inicializador

Classe Cao em Python

```
class Mamifero(object):  
    """lição de casa: implementar"""  
  
class Cao(Mamifero):  
    qt_patas = 4  
    carnivoros = True  
    nervoso = False  
    def __init__(self, nome):  
        self.nome = nome  
    def latir(self, vezes=1):  
        # quando nervoso, late o dobro  
        vezes = vezes + (self.nervoso * vezes)  
        print self.nome + ':' + ' Au!' * vezes  
    def __str__(self):  
        return self.nome  
    def __repr__(self):  
        return 'Cao({0!r})'.format(self.nome)
```



- self é o 1º parâmetro em todos os métodos de instância

- atributos da instância só podem ser acessados via self

Classe Cao

```
class Mamifero(object):  
    """lição de casa: implementar"""  
  
class Cao(Mamifero):  
    qt_patas = 4  
    carnivoro = True  
    nervoso = False  
    def __init__(self, nome):  
        self.nome = nome  
    def latir(self, vezes=1):  
        # quando nervoso, late o dobro  
        vezes = vezes + (self.nervoso * vezes)  
        print self.nome + ':' + ' Au!' * vezes  
    def __str__(self):  
        return self.nome  
    def __repr__(self):  
        return 'Cao({0!r})'.format(self.nome)
```

invocação

```
>>> rex = Cao('Rex')  
>>> rex  
Cao('Rex')  
>>> print rex  
Rex  
>>> rex.qt_patas  
4  
>>> rex.latir()  
Rex: Au!  
>>> rex.latir(2)  
Rex: Au! Au!
```

• na invocação do método, a instância é passada implicitamente na posição do self

Doctests

- Um dos módulos para fazer testes automatizados na biblioteca padrão de Python
 - o outro módulo é o unittest, da família xUnit
- Doctests foram criados para testar exemplos embutidos na documentação
- Usaremos doctests para especificar exercícios
- Exemplo: `$ python -m doctest cao_test.rst`