Programando em Python — Aula 1

Gustavo Sverzut Barbieri <barbieri@gmail.com>

Sumário

1 Introdução

Esta aula é parte de uma apostila introdutória à linguagem de programação Python. Supõe-se que o leitor seja um programador em alguma outra linguagem, isto é, não abordarei conceitos básicos de programação.

Esta primeira aula pode ser utilizada como um manual de referência rápida. Ela contém tudo que você precisa saber para programar em Python.

2 Por que Python

Python é uma linguagem fácil e gostosa de ser utilizada. É intuitiva e quem aprende uma vez nunca se esquece pois é muito próxima de um pseudo-código. Ela não fica no seu caminho na hora de programar.

Python é muito fácil de ser ensinada, pois ela foi elaborada por Guido Van Rossum em 1991 para o ensino de programação. Ela é orientada a objetos sem lhe forçar a programar desta maneira. Não reinventa funções amplamentes conhecidas no mundo C/C++, como por exemplo, para abrir um arquivo utiliza-se `open( file )` e não `new FileInputStream( new File( file ) )` como em Java.

Python tem uma vasta biblioteca de funcionalidades incluída que consiste de implementações diversas chegando até ao manuseio de conexões HTTP seguras, processamento de XML e HTML, bancos de dados e muito mais. Com módulos extras para coisas mais específicas como "engines" de jogos e processamento de imagens. Possui módulos para chamadas entre-processos com RPC, RPC-XML e outros, incluindo até o RMI[1].

Existe uma grande comunidade em volta (http://www.python.org, http://www.pythonbrasil.com.br) e é amplamente utilizada nos meios acadêmico, software livre, pesquisa (Google, NASA), jogos (Disney) dentre outros.

Existe uma palestra cujo tema é chamar programadores Java para o mundo Python do Osvaldo Santana Neto em http://www.pythonbrasil.com.br/OsvaldoSantanaNeto.

Algums comentários de "gente grande" podem ser vistos em http://www.python.org/Quotes.html.

Para mais informações, inicie o python e importe o módulo "this", seque abaixo o conteúdo:

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

[1] este só para a implementação Jython, que roda dentro de uma máquina virtual Java

3 Hello World

Tudo começa com um *hello world*, e nossa aula não é diferente.

Como Python é uma linguagem interpretada, podemos executar um programa interativamente, sendo um ótimo meio para fazer cálculos avançados sem precisar abrir uma planilha de cálculo ou até mesmo fazer protótipos rápidos. Nosso primeiro código será direto no interpretador. Primeiro, inicie o interpretador Python, geralmente é só chamar **python**. Ele deverá mostrar algo assim na sua tela:

```
[gustavo@gustavo] ~/aulas_python/src$ python
Python 2.3.3a0 (#2, Nov 20 2003, 07:51:52)
[GCC 3.3.2 (Debian)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
```

Logo após já pode entrar com o código. Para um simples *Hello World*:

```
print "hello world!"
```

Caso você queira o código em um arquivo, para que não precise digitar toda vez, salve o código acima em um arquivo e depois execute: `python arquivo_que_você_salvou_o_código`. Se você quiser gerar um executável, é só salvar o seguinte código em um arquivo e dar permissões de execução:

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

print "hello world!"
```

No qual a primeira linha é um comentário, comentários em Python começam com # e se estende até o final da linha.

Nota:

No caso da primeira linha temos algo comum em ambientes Unix, que são os comentários funcionais. Se na primeira linha de um arquivo existir #! seguido de um caminho para um executável, este executável será chamado e o nome do arquivo será passado como argumento. No nosso caso, o arquivo é hello_world.py e o Unix vai identificar a primeira linha como sendo um comentário funcional e executar da seguinte forma:

```
/usr/bin/env python hello_world.py
```

Na segunda linha temos outro comentário funcional, porém desta vez ele é usado pelo Python (e não pelo Unix, como visto na nota acima). Este comentário funcional especifica em qual codificação foi escrito o código em *hello_world.py* e tem o intuito de facilitar pessoas de outras línguas a escreverem código Python (chineses, japoneses e outros podem escrever as mensagens de texto em linguagem nativa). Este comentário funcional é utilizado a partir do Python 2.3.

Na terceira linha temos uma função interna do Python que imprime na tela, a função `print`. Você pode encontrar a documentação em http://www.python.org/doc/current/ref/print.html.

De agora em diante você já sabe como executar códigos Python. Fica a seu critério executar os exemplos aqui listados em modo interativo ou via arquivos.

Nota:

Se você quiser que o interpretador interativo complete o código você pode usar um recurso não documentado, basta digitar:

```
import rlcompleter
import readline
readline.parse_and_bind("tab: complete")
```

Isso só vai funcionar se você possuir o suporte ao readline compilado, o que deve ser verdade na maioria das instalações GNU/Linux.
Auto-completar o código fonte como o Eclipse faz para o Java é impossível para a maioria dos códigos Python por que a tipagem é dinâmica e o editor de textos não tem como saber o objeto que vai estar presente na execução. Porém como estamos no interpretador este sabe qual o objeto e portanto consegue completar como o Eclipse e outros.

3.1 A função `print`

Esta função recebe uma *string*[2] e a coloca em uma nova linha da saída padrão[3], isto é, após o texto dado ela envia um caractere de controle ($\backslash n$) para o terminal de saída avançar para a próxima linha.

Caso queiramos suprimir este caracter de controle de nova linha, é só colocar uma vírgula no fim do comando, veja:

```
print "Texto SEM nova linha." ,
print "Texto COM nova linha."
```

Isto resultará em:

```
Texto SEM nova linha.Texto COM nova linha
```

A função `print` utiliza-se de um recurso das *strings* para simular o **`printf`** do C, C++, PHP e outras. Repetimos: este recurso **é das *strings*** (seria o **`sprintf`** de C), portanto pode ser utilizado em outros lugares. Ele tem a forma:

```
"STRING FORMATO" % ( LISTA DE VALORES )
```

No qual `"STRING FORMATO"` segue o padrão do **`printf`**, com **`%s`** para *strings*, **`%f`** para números reais (ponto flutuante), **`%d`** para decimais. Vide manual do Python para mais informações. Uma das principais diferenças é que o **`%s`** é mais relaxado, aceitando todos os tipos de conteúdo e convertendo-os para texto.

Exemplo:

```
minha_string = "a=%d, b=%x" % ( 1234, 0xff00ff )
print "decimal: '%d', flutuante: '%.2f'" % ( 1234, 1.234 )
print "string: '%s', string: '%s'" % ( "teste", 123 )
```

4 Variáveis

Em Python uma variável não tem tipo fixo, apenas o tipo do conteúdo atual. Elas podem ser inicializadas em qualquer parte do código — porém deve-se **tomar extremo cuidado para não utilizar uma variável antes desta ser sido inicializada!** Os nomes de variáveis só podem ter letras de **a** a **z**, o caractere de sublinhado, "_", e números. Letras maiúsculas e minúsculas são consideradas diferentes! Para incializar uma variável é só atribuir um valor a ela, por exemplo:

[2] *string* pode ser entendido como um pedaço de texto, uma cadeia de caracteres.
[3] O termo saída padrão é comumente referido por `stdout`, do inglês *standard output*.

```
texto1  = "algum texto"
texto2  = 'outro texto'
inteiro = 123
real    = 1.23
real2   = 1.23e6   # 1.23 * 10^6

lista = [ 1, 2, "teste", 1.23, inteiro ]

print "lista=%s" % lista # imprime: lista=[ 1, 2, 'teste', 1.23, 123 ]
print "lista[0]=%s, lista[1]=%s" % ( lista[ 0 ], lista[ 1 ] )
# imprime: lista[0]=1, lista[1]=2
print "pedaço da lista: %s" % lista[ 1 : 3 ]
# imprime: pedaço da lista: [ 2, 'teste' ]


dicionario = { "chave"   : "valor",
               "inteiro" : inteiro,
                inteiro  : "inteiro" }

print "dicionario=%s" % dicionario
# imprime: dicionario={'chave': 'valor', 'inteiro': 123, 123: 'inteiro'}
print "dicionario['chave']=%s" % dicionario["chave"]
# imprime: dicionario["chave"]=valor
print "dicionario[123]=%s" % dicionario[123] # inteiro == 123
# imprime: dicionario[123]=inteiro
print "dicionario['chave']=%(chave)s" % dicionario
```

5 Usando Condicionais: `if ...elif ...else`

Os elementos condicionais em Python funcionam tais como em outras linguagens, com a sintaxe:

```
if CONDIÇÃO:
     BLOCO DE CÓDIGO
elif CONDIÇÃO:
     BLOCO DE CÓDIGO
else:
     BLOCO DE CÓDIGO
```

veja um exemplo:

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

variavel1 = 1
variavel2 = 2

if variavel1 == 1:
    print "variavel1 igual a 1"
elif variavel1 !=1 or not variavel2 == 2:
    print "variavel1 diferente de 1 ou variavel2 diferente de 2"
elif variavel1 >= 1000 or variavel2 <= -1000:
    print "variavel1 maior que 1000 ou variável2 menor que -1000"
else:
    print "nenhuma das alternativas anteriores"
```

Nota-se que as condições podem se utilizar de operadores `==` (igual), `!=` (diferente), `<` (menor), `>` (maior), `<=` (menor ou igual), `>=` (maior ou igual) e que várias condições podem ser combinadas em uma condição maior utilizando os operadores lógicos `and` (e), `or` (ou) e `not` (negação).

> **Atenção:**
>
> *Em Python os blocos de comandos são definidos pela endentação (espaços deixados antes do início do texto), portanto é muito importante ficar atento ao programar!* Muitos dos erros advém de má endentação.

6 Laços: `while` e `for`

A linguagem provê dois tipos de laços, um que testa uma condição e executa o bloco de código enquanto tal for verdadeira (`while`) e outro que interage com uma seqüência (`for`). As sintaxes são:

```
while CONDIÇÃO:
    BLOCO DE CÓDIGO
```

```
for VARIAVEL in SEQÜÊNCIA:
    BLOCO DE CÓDIGO
```

Exemplo de laços:

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

for fruta in [ "banana", "maçã", "uva" ]:
    print "Fruta: " + fruta
# end for fruta

for i in range( 0, 10 ):
    print "i = " + str( i )
# end for i

i = 0
while i < 10:
    print "i = %d" % i
    i += 1
# end while i
```

Dentro de laços é possível usar dois comandos adicionais para ajudar no controle dos mesmos. Os comandos são:

- `continue` este comando vai para a próxima iteração do laço imediatamente.
- `break` este comando interrompe a execução do laço.

No exemplo utilizamos um comentário logo após o término de um laço, isto não é obrigatório porém ajuda a prevenir erros de endentação. Achamos uma boa prática e aconselhamos que utilizem-a.

7 Funções

Funções são um método de reaproveitamento e organização de código. Uma função pode receber alguns parâmetros e devolver um resultado.

Uma função é chamada pelo seu nome, seguido de parênteses contendo os parâmetros.

No exemplo sobre laços (Listagem 6) `range( 0, 10 )` é uma função chamada `range`, que recebe como parâmetros dois números inteiros, no exemplo passamos os valores 0 e 10. Esta função também retorna um valor, que é uma seqüência de números desde 0 até 10 (não incluindo este último).

Em Python tudo são objetos, incluindo as funções. Elas tem alguns atributos, dentre os quais os mais úteis são `__doc__` o qual contém a documentação da função e `__name__` que contém o nome da função. Mas espera, pra que eu quero saber o nome da função se eu tenho que chamar ela pelo nome? Bem, você pode mudar o nome da função, por exemplo, se uma função tiver um nome muito grande, pra economizar digitação, você pode criar um novo nome apontando pra função que você quer (lembra-se que em Python tudo é um objeto?). Exemplo:

```
r = range
print r( 10 ) == range( 10 )
```

Para ver os atributos de um objeto (no caso, de uma função) utiliza-se a função `dir( OBJETO )`. Exemplo:

```
print dir( range )
for atributo in dir( range ):
      print "%s = %s" % ( atributo, getattr( range, atributo ) )
# end for atributo
```

Neste exemplo, `getattr` é uma função que recebe como primeiro parâmetro o objeto e como segundo o atributo que se deseja receber o valor, é equivalente a fazer `objeto.atributo`. Outra peculiaridade que vemos no exemplo anterior é que não importa o tipo do conteúdo dos atributos, eles são convertidos para *strings* pelo uso do `%s`, isto é feito pelo Python usando a função `str( OBJETO )`.

7.1 Criando Funções

Agora que você sabe usar funções, deve saber também como criar suas próprias. Trataremos só do básico, porém aconselhamos que você leia o manual do Python ou outros tutoriais mais avançados para saber os recursos disponíveis.

A sintaxe de criar uma função é simples:

```
def NOME DA FUNÇÃO( LISTA DE PARÂMETROS ):
    """
    DOCUMENTAÇÃO DA FUNÇÃO
    """
    BLOCO DE CÓDIGO
```

Onde `NOME DA FUNÇÃO` deve ser um nome com apenas caracteres de **a** a **z** (letras maiúsculas e minúsculas são consideradas diferentes!), o caractere de sublinhado, "_", e números, `LISTA DE PARÂMETROS` é uma lista de nomes de variáveis separadas por vírgulas, tais variáveis estarão disponíveis no escopo da função com os valores recebidos.

O texto entre a seqüência de três aspas serve para documentar a função e chama-se *"docstring"*. Existem diversos aplicativos que geram a documentação em vários formatos (html, man, pdf, ...) a partir das *docstrings*. Também podemos acessar a documentação de uma função lendo o conteúdo do atributo `__doc__` da função (Vide exemplo, Listagem 7.1). Você também pode usar a função `help( função )` para obter a documentação desta.

O bloco de código pode conter várias saídas (pontos de retorno), basta somente usar o comando `return VALOR A RETORNAR` ou apenas `return`, caso não queira retornar valor algum.

Exemplo:

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

def fatorial( numero ):
        """
        Função recursiva que retorna o valor do fatorial do número dado.
        """

        if numero <= 1:
                return 1
        else:
                return ( numero * fatorial( numero - 1 ) )
# end fatorial

for n in range( 1, 11 ):
        print "Fatorial de %d é %d" % ( n, fatorial( n ) )
# end for n

print "\nDocumentação da função fatorial:\n" + fatorial.__doc__
```

Importante:

O Python passa como parâmetro para a função uma ***cópia da referência para o objeto****, por isso se você mudar a referência para outro objeto, a variável passada como parâmetro não será alterada, caso você alterar o conteúdo de um objeto, a variável terá seu valor alterado. Um exemplo facilita o entendimento:*

```
lista = [ 1, 2, 3 ]

def nao_altera_lista( l ):
    l = [ 11, 22, 33 ]       # Muda a referência para outro
                             # objeto. Portanto não altera o
                             # conteúdo de "lista"

nao_altera_lista( lista )
print lista                  # imprime: [ 1, 2, 3 ]

def altera_lista( l ):
    l.append( 123 )          # Altera o conteúdo do objeto.
                             # Portanto altera o conteúdo de
                             # "lista"

altera_lista( lista )
print lista                  # imprime: [ 1, 2, 3, 123 ]
```

7.2 Funções com Argumentos Pré-Definidos

Pode-se utilizar argumentos com valores pré-definidos, os quais, se não forem especificados pelo usuário, assumirão o dado valor. Veja o exemplo:

```
def f( a, b=1 ):
    print "a='%s'   b='%s'" % ( a, b )
```

```
f( 2, 2 )   # imprime  a='2'   b='2'
f( 2 )      # imrpime  a='2'   b='1'
```

É possível fornecer um argumento específico sem que seja necessário especificar todos apenas nomeando o valor:

```
def f( a, b=1, c=2, d=3 ):
    print "a='%s'   b='%s'   c='%s'   d='%s'" % ( a, b, c, d )

f( 2, 2 )          # imprime  a='2'   b='2'   c='2'   d='3'
f( 2 )             # imprime  a='2'   b='1'   c='2'   d='3'
f( 2, c=4 )        # imprime  a='2'   b='1'   c='4'   d='3'
f( 2, c=4, d=5 )   # imprime  a='2'   b='1'   c='4'   d='5'
```

Importante:

Ao definir valores padrão para argumentos, nunca utilize listas, dicionários ou classes. Utilize apenas tipos imutáveis: números, strings, tuplas ou o valor `None`.
Isto ocorre pois o objeto referido na descrição da classe ou o valor padrão do método é instanciado na hora da definição e não na hora da execução do método.

```
# Implementação "errônea" (Vide texto explicativo abaixo)
def f( arg=[] ): # Errado!!!
    return arg

a = f()
b = f()

print a # imprime []
print b # imprime []

a += [ 1 ]
print a # imprime [1]
print b # imprime [1] também!!!


# Implementação correta:
def f( arg=None ):
    a = arg or [] # Mesmo que: 'if arg: a=arg else a=[]'
    return a

a = f()
b = f()

print a # imprime []
print b # imprime []

a += [ 1 ]
print a # imprime [1]
print b # imprime []
```

Note que a implementação acima ***não está errada*** *(por isso as aspas na palavra errônea). É apenas um recurso da linguagem que você tem que saber usar. Em geral queremos que o parâmetro seja algo novo, que possa ser usado como no exemplo. Porém, certas vezes deseja-se usar uma única instância para todas as chamadas e seria correto usar essa instância como valor, obtendo-se o funcionamento desejado (como para "singletons"). O mesmo vale para os atributos de classe versus atributos de instância.*

7.3 Funções com Número Variável de Argumentos

Pode-se utilizar funções que recebem um número indefinido de argumentos. Estes podem ser nomeados ou não. Tudo fica mais fácil com exemplos:

```
def arg_sem_nome( *args ):
    for arg in args: # args é uma lista com os valores
        print "arg: %s" % arg

arg_sem_nome( "a", "b", 123, 1.23 )
# imprime:
```

```
# arg: a
# arg: b
# arg: 123
# arg: 1.23
```

```
def arg_com_nome( **args ):
    for arg in args: # args é um dicionário com as chaves/valores
        print "%s=%s" % ( arg, args[ arg ] )

arg_com_nome( a=123, b=4.56, teste="teste12345" )
# imprime:
# a=123
# b=4.56
# teste=teste12345
```

8 Classes

A implementação de classes em Python é extremamente fácil e suporta herança múltipla, possui troca de operadores, porém não tem sobrecarga de métodos.

8.1 Criando Uma Classe

Para criar uma classe utiliza-se a sintaxe:

```
class MINHA_CLASSE( CLASSE_MAE_1, CLASSE_MAE_2, ... ):
    """
    DOCUMENTACAO MINHA_CLASSE
    """

    ATRIBUTO_1      = VALOR_ATRIBUTO_1
    ATRIBUTO_2      = VALOR_ATRIBUTO_2
    __ATRIBUTO_3__ = VALOR_ATRIBUTO_3
    __PRIVADO_1     = VALOR_PRIVADO_1
    __PRIVADO_2_    = VALOR_PRIVADO_2

    def __init__( self, OUTROS_PARAMETROS ):
        """DOCUMENTAÇÃO DO CONSTRUTOR"""
        BLOCO DE CÓDIGO DO CONSTRUTOR

    def __del__( self ):
        """DOCUMENTAÇÃO DO DESTRUTOR"""
        BLOCO DE CÓDIGO DO DESTRUTOR

    def METODO_1( self, PARAM_1, PARAM_2=VALOR_PADRAO ):
        BLOCO DE CÓDIGO

    def __METODO_PRIVADO_1( self, PARAM_1 ):
        BLOCO DE CÓDIGO
```

Algumas peculiaridades da definição acima:

- Construtores são indicados por `__init__()` e destrutores por `__del__()`. Estes são operadores.
- Atributos e métodos iniciados por dois sublinhados "__" e terminados com no máximo um sublinhado são privados. Sim, o nível de proteção de um atributo/método é baseado no nome dele! `__PRIVADO_1`

e `__PRIVADO_2_` são atributos privados, enquanto que `ATRIBUTO_1`, `ATRIBUTO_2` e `__ATRIBUTO_3__` são públicos. Isto pode parecer estranho, mas é muito útil ao programar, pois baseado no nome você já sabe se algo é público ou privado.

- Todos os métodos de instância tem que receber o primeiro parâmetro como sendo o objeto representante da própria classe. Em outras linguagens isto é feito pelo compilador, em python isto é explícito. Por convenção chamamos de `self`, porém se você estiver acostumado com Java/C++ pode chamar de `this` ou qualquer coisa que lhe agradar mais.

Muita coisa lhe pareceu estranha? Não se preocupe... é mais fácil quando se faz algo real, portanto vamos a um exemplo:

```
class Pessoa:

    def __init__( self, nome="", idade=0 ):
        """
        Cria uma pessoa.
        Parâmetros:
           - nome é opcional (padrão: "").
           - idade é opcional (padrão: 0).
        """
        self.nome  = nome
        self.idade = idade
    # __init__()

    def getIdade( self ):
        """Pega a idade da pessoa"""
        return self.idade
    # getIdade()

# Usando:
p1 = Pessoa()                 # Nova classe
p2 = Pessoa( "Gustavo" )      # Nova classe, passando o nome
p3 = Pessoa( "Gustavo", 22 ) # Nova classe, passando o nome e idade

p1.nome  = "Teste"  # Atribui nome a p1
p2.idade = 22       # Atribui idade a p2

i = p3.getIdade()   # Utiliza método getIdade() de p3
```

Neste exemplo foi definido uma classe Pessoa e instanciados três objetos: p1, p2 e p3.

Importante:

Como mostra `getIdade()`*, sempre que utilizar um atributo da classe, precisa utilizar o* `self`*, ou qualquer nome definido como primeiro elemento do método, isto é devido à política do python:*

Explicit is better than implicit.

Isto mantém sempre as coisas claras. Você sempre vai especificar de onde o valor vem.

Importante:

Cuidado para não confundir variáveis de classe e variáveis de instância. Ambas podem ser acessadas através de uma referência para uma instância.

```
class A:
    a = [] # Variável de classe
    def __init__( self, b={}, d="teste" ):
        self.b = [] # Variável de instância


obj1 = A()
obj2 = A()
obj1.a.append( 1 )

print obj1.a          # imprime [1]
print obj2.a          # imprime [1]
print A.a             # imprime [1] também!!

obj2.b.append( 1 )
print obj1.b          # imprime []
print obj2.b          # imprime [1]
print A.b             # gera erro: "class A has no attribute 'b'"
```

Usos para as variáveis de classe (a no exemplo) são diversos, como criar "singletons" ou mesmo armazenar um semáforo para todas as instâncias da classe (o que não deixa de ser um "singleton" do semáforo).
Porém tenha cuidado ao implementar, verifique se é este o comportamento que precisa.

```
class PessoaRG( Pessoa ):
    def __init__( self, RG, nome="", idade=0 ):
       Pessoa.__init__( self, nome, idade ) # inicia nome e idade
       self.RG    = RG
```

Neste exemplo mostramos como criar uma nova classe PessoaRG herdando da classe Pessoa. Para esta nova classe criamos um novo construtor que acrescenta o parâmetro RG e utiliza o construtor da classe Pessoa para iniciar nome e idade. Note que o funcionamento de `Pessoa.__init__()` seria equivalente ao super() de Java, porém isso é mais explícito.

O acesso a `Pessoa.__init__()` também nos mostra que é possível acessar os métodos e atributos de uma classe sem ter que instanciá-la, um exemplo:

```
class A:
    a = 1234

print A.a # imprime 1234

# Porém para utilizar os métodos precisa ser descendente:
a = A()
Pessoa.__init__( a, "Teste ERRADO", 1234 )
# Gera o erro:
#  Traceback (most recent call last):
#    File "<stdin>", line 1, in ?
#  TypeError: unbound method __init__() must be called with Pessoa
#             instance as first argument (got A instance instead)
```

```
# Correto seria:
class A( Pessoa ):
    a = 1234

print A.a # imprime 1234
a = A()
Pessoa.__init__( a, "Teste CORRETO", 12 )
print a.nome  # imprime Teste CORRETO
print a.idade # imprime 12
print a.a     # imprime 1234
```

Nota:

Note a simplicidade e elegância do Python. Ao utilizar `self` *como primeiro parâmetro é possível chamar qualquer método de uma função passando um objeto para ela. Além disso a implementação do interpretador fica fácil e ainda permite que se utilize os conceitos básicos de herança múltipla, pois fica a cargo do usuário decidir o construtor a usar.*

8.2 Controlando o Acesso aos Recursos

Como vimos anteriormente os recursos são controlados pelo nome do mesmo. Aqui faremos alguns testes para demonstrar o funcionamento:

```
class A:
    a   = 1
    __b = 2

class B( A ):
    __c = 3
    def __init__( self ):
        print self.a
        print self.__c
        print self.__b # Vai dar erro, pois __b não existe em 'B'!

a = A()
print a.a   # imprime 1
print b.__b # erro, __b é restrito a 'A'

b = B() # erro, vide acima (B.__init__() não tem acesso a __b)
```

Outra forma de controlar o acesso aos atributos é especificar as funções de consultar, escrever e apagar (*get*, *set* e *del*) dos atributos de forma individual. Para isso a classe **precisa** herdar de `object` — isto é, precisa ser uma classe do novo estilo ou *"new style class"*. O controle pode ser feito usando os métodos `__getattribute__()`, `__getattr__()`, `__setattr__()` e `__delete` ou então a facilidade `property()`:

```
class A:
    def __init__( self ):
        self.__a1__ = 123

    def __set_a1__( self, value ):
       print "Atribuindo valor: '%s' a a1." % value
       self.__a1__ = value
```

```
    def __get_a1__( self ):
       print "Obtendo valor de a1."
       return self.__a1__

    def __del_a1__( self ):
       print "Apagando a1."
       del self.__a1__

    a1 = property( __get_a1__, __set_a1__, __del_a1__ )

a = A()
print a.a1  # Imprime "Obtendo valor de a1." e depois: "123"
a.a1 = 222  # Imprime "Atribuindo valor: '222' a a1."
del a.a1    # Imprime "Apagando a1."
```

O acesso de leitura ao atributo `a1` será controlado por `__get_a1__()`, escrita a `__set_a1__()` e a ação de apagar a `__del_a1__`, com isso pode-se limitar tais ações. Ao passar `None` para qualquer um dos parâmetros de `property()` tal ação será proibida.

8.3 Herança Múltipla

O suporte a herança múltipla é bem básico. É possível utilizar-se deste recurso, porém ele tem algumas limitações, que exploraremos a seguir:

```
class A( object ):
   a = 1

class B( object ):
   a = 2
   b = 2

class C( A, B ):
   pass # Não tem corpo. Usado para permitir bloco vazio

class D( B, A ):
   pass # Não tem corpo. Usado para permitir bloco vazio

a = A()
b = B()
c = C()
d = D()

print a.a # imprime 1

print b.a # imprime 2
print b.b # imrpime 2

print c.a # imprime 1  <---
print c.b # imrpime 2

print d.a # imprime 2  <---
print d.b # imprime 2

print D.__mro__ # imprime ordem de percurso na hierarquia de classes
                # (<class '__main__.D'>, <class '__main__.B'>,
                # <class '__main__.A'>, <type 'object'>)
```

```
print C.__mro__ # imprime (<class '__main__.C'>, <class '__main__.A'>,
                #<class '__main__.B'>, <type 'object'>)
```

Como vimos, a herança múltipla é implementada buscando por atributos definidos nas classes mãe da esquerda para a direita, parando quando o mesmo é encontrado.

Nota:

Neste exemplo as classes herdam de `object`*. Classes que herdam deste são chamadas de "novo estilo" (new style classes) e fornecem uma gama de funcionalidades extras, como o atributo* `__mro__`*, a funcionalidade* `property()`*, os métodos* `__setattr__()`*,* `__getattr__()`*,* `__delattr__()`*, dentre outros.*

8.4 Simulando Sobrecarga de Métodos

Python não suporta sobrecarga de métodos, ao definir um método com um nome já definido, o anterior é trocado pelo mais recente. Por exemplo:

```
class A:
   def a( self, a ):
       print a
   def a( self, a, b ):
       print "%s, %s" % ( a, b )

a = A()
a.a( "teste" )       # Não funciona... ele requer 3 argumentos (self + 2)
a.b( "teste", 123 ) # Funciona perfeitamente
```

Como a linguagem suporta parâmetros variáveis e os nomes de variáveis não tem tipos (não é tipagem estática, em tempo de compilação), isto é resolvido de várias formas dependendo da necessidade:

- **Tratar tipos diferentes de parâmetros:** devido ao python não atribuir tipos aos nomes das variáveis, a implementação fica trivial, trata-se os parâmetros passados usando `isinstance( parametro, classe )` e então toma-se a ação:

```
class A:
    def m( self, a ):
        if isinstance( a, int ):
            print "inteiro"
        elif isinstance( a, str ):
            print "string"
        else
            raise TypeError, "Tipo %s não suportado." % a.__class__.__name__
```

Importante:

Evite usar este tipo de programação!
O interessante em Python é ele ser de tipagem dinâmica e você deve aproveitar isso. ***Não confira o tipo, confira a interface!*** *Isto é, verifique se o objeto implementa os métodos necessários para o funcionamento do algoritmo.*
Por exemplo, existe algum algoritmo de ordenação para inteiros. Ora, por que não ordenar textos, listas, tuplas ou qualquer outra lista de objetos com o mesmo código? Se tudo que você precisa para ordenar uma lista de objetos é conseguir compará-los, qualquer um que implemente os métodos necessários (`__le__()`, `__ge__()`, `__eq__()`, ...), funcionará!
Para testar a interface pode-se utilizar `hasattr()` e `callable()`:

```
def f( p ):
   if not ( hasattr( p, "metodo_necessario" ) and \
            callable( p.metodo_necessario ) ):
      raise TypeError, "p precisa implementar metodo_necessario()"

   codigo_da_funcao_usando p.metodo_necessario
```

Um uso de `isinstance()` é com objetos que devem herdar de uma classe que não implementa código, apenas define os métodos, também conhecida como "interface". Porém isso traz alguns problemas também: o primeiro é que a classe herdeira pode definir um atributo com o mesmo nome do método, ela vai continuar sendo válida para a conferência, porém não vai funcionar (daí a necessidade da chamada ao `callable()` na conferência acima); outro problema é existir algum código na interface (não deveria), se este possuir um problema ele será propagado.

Vide "isinstance() considered harmful" (http://www.canonical.org/~kragen/isinstance/) o qual explica vários problemas em usar `isinstance()`.
Para um projeto maior, algo já pronto é recomendado, vide o PyProtocols em http://peak.telecommunity.com/PyProtocols.html.

Outra maneira seria utilizar `try ... except` para os casos que os parâmetros tem interface similares e capturar os que fogem à regra.

- **Tratar número diferente de parâmetros:** pode-se utilizar os mesmos recursos de funções com parâmetros variáveis ou então parâmetros com valores padrão.

8.5 Reimplementando Operadores

Em Python pode-se reimplementar operadores. Os operadores na verdade chamam métodos específicos. Como visto acima, para trocar um método é só reimplementar o mesmo. O Python provê uma interface básica de operadores que podem ser implementados. Pode-se ver os métodos que uma classe/objeto implementa com o comando `dir()`, os operadores são nomeados com `__` no início e fim do nome. Os mais interessantes são:

- `+`, `__add__( self, outros )` implementa o símbolo `+` como operador binário. É muito utilizado para juntar duas coisas (listas, tuplas, números, ...).
- `str()`, `__str__( self )` retornar uma string representando o objeto. É muito utilizado para impressão de objetos na tela. Se precisar customizar a apresentação da sua classe, utilize este. Também pode ser utilizado para mapear uma classe para XML e ao se imprimir a classe, esta é feita de modo a formar uma estrutura de tags.
- `[x]`, `__getitem__( self, x )` pega o elemento de índice x.

- `[x]=y`, `__setitem__( self, x, y )` atribui y ao valor do elemento de índice x.
- `==`, `__eq__( self, outro )` testa a igualdade.

Exemplo:

```
class HTML_A:
    href  = ""
    cdata = ""
    def __str__( self ):
        return '<a href="%s">%s</a>' % ( self.href, self.cdata )

a = HTML_A()
a.href  = "http://www.unicamp.br"
a.cdata = "Site da Unicamp"

print a # Imprime <a href="http://www.unicamp.br">Site da Unicamp</a>
```

8.6 Exceções

Python implementa exceções como sendo classes. Uma exceção pode herdar da classe base `Exception` ganhando assim toda a funcionalidade desta.

Uma exceção pode ser capturada com um bloco `try: COMANDOS except TIPO_EXCEÇÃO, EXCEÇÃO: COMANDOS`

```
try:
    a[ key ] = value
except KeyError, e: # Captura exceção do tipo KeyError, guarda em 'e'
    print "O valor usado como índice (%s) não existe: %s" % ( key, e )
except TypeError:   # Captura exceção do tipo TypError, descarta-a
    print "Tipo errado!"
except:             # Captura qualquer outra exceção
    pass # Não faz nada, ignora.
```

Em alguns casos não queremos tratar as exceções, porém desejamos que alguns comandos sejam executados antes de prosseguir com a excessão para o próximo nível. Para isso utilizamos o bloco `try: COMANDOS finally: COMANDOS`. Exemplo:

```
try:
    f = open( "arquivo.txt", "r" )
    a[ key ] = value
finally:
    f.close() # Sempre executa o f.close()
```

Nota:

Não é possível utilizar `try: ... except: ... finally: ...`! Você tem que aninhá-los ou apenas usar o `except:`, que é o que é feito na maioria das vezes.

9 Módulos

Para utilizar módulos é bem fácil, apenas importe o módulo que você pretende utilizar. Por exemplo:

```
import urllib
```

Isto carregará o módulo urllib, que proporciona uma interface de conexão e tratamento de URLs. Note porém que toda funcionalidade está dentro **name space** (espaço de nomes) urllib e portanto todo acesso deve começar com ele:

```
import urllib
url = urllib.urlopen( "http://www.ic.unicamp.br" )
conteudo = url.read()  # Notou a facilidade de se downloadear uma página???
url.close()
```

Algumas vezes queremos os símbolos no mesmo espaço de nomes que nosso aplicativo, neste caso podemos importá-los, todos ou seletivamente:

```
from urllib import urlopen
url = urlopen( "http://www.ic.unicamp.br" )
conteudo = url.read()
url.close()
print urllib.quote( "teste?teste&bla" )
```

```
from urllib import *
url = urlopen( "http://www.ic.unicamp.br" )
conteudo = url.read()
url.close()
print quote( "teste?teste&bla" )
```

Para criar um módulo é simples, crie um arquivo terminado em `.py` e coloque-o no seu diretório corrente, no diretório com suas bibliotecas python (ié: `/usr/lib/python2.3/site-packages/`) ou em algum lugar que a variável de ambiente `PYTHONPATH` aponte.

Uma alternativa é construir um diretório e nele colocar os arquivos com os submódulos.

Procure pelos módulos disponíveis em:

- http://www.python.org/doc/current/lib/lib.html Módulos disponíveis na biblioteca padrão.
- http://www.python.org/pypi Python Package Index.

10 Encerramento

Com o conteúdo desta aula você deve ter aprendido o básico do Python. Já é possível fazer aplicativos completos, porém pretendo desenvolver algumas aulas mais para passar algumas dicas desta linguagem que eu adoro. Existem muitos recursos para facilitar sua vida! Se você não descobrir sozinho, aguarde por um próximo texto!

11 Referências

- http://www.python.org Site oficial.
- http://www.pythonbrasil.com.br Site da comunidade brasileira.
- http://www.python.org/pypi Lista de módulos extra.

12 Resumo: Variáveis e Controle de Fluxo

Variáveis e Substituição:

```
lista = [ 1, 2, "texto", 3.5 ]
print lista[ 0 ]      # imprime 1
print lista[ 1 : 2 ] # imprime [ 2, "texto" ]
print lista[ : -1 ]  # imprime [ 1, 2, "texto" ]
lista += [ "novo" ]
print lista           # imprime [ 1, 2, "texto", 3.5, "novo" ]

tupla = ( 1, 2, "texto", 3.5 ) # Elementos não podem ser alterados!
print tupla[ 0 ]      # imprime 1
print tupla[ 1 : 2 ] # imprime ( 2, "texto" )
print tupla[ : -1 ]  # imprime ( 1, 2, "texto" )
tupla += ( "novo", )
print tupla           # imprime ( 1, 2, "texto", 3.5, "novo" )

dicionario = { "chave": "valor", "c2": "v2" }
print dicionario[ "chave" ] # imprime valor

newstring1 = "string='%s' int='%d' float='%03.2f'" % ( "txt", 123, 4.56 )
newstring2 = "chave=%(chave)s  c2=%(c2)s" % dicionario
newstring3 = "chave=%s  c2=%s" % ( dicionario[ "chave" ], dicionario[ "c2" ] )
```

Controle de Fluxo e Laços:

```
if a > b and a < c:
    print "a entre b e c"
elif a > c:
    print "a maior que c"
else:
    print "a menor ou igual a b ou igual a c"

for elemento in lista:
    print "elemento: %s" % elemento

coordenadas = [ ( 0, 0, 0 ), ( 1, 0, 0 ), ( 0, 1, 0 ), ( 0, 0, 1 ) ]
for x, y, z in coordenadas:
    print "Ponto: x=%d, y=%d, z=%d" % ( x, y, z )

loop = 1
while loop:
    resultado = faca_acao()
    if resultado < 0:
       break    # Pára o laço
    else resultado > 0:
       continue # Volta para o começo do laço
    print "teste"
```

13 Resumo: Funções e Classes

Funções:

```
def funcao( p1, p2="Valor Padrao" ):
    print "p1: '%s'  p2: '%s'" % ( p1, p2 )

def f_param_variaveis( p1, *args ):
    print "p1: '%s'" % p1
    for arg in args:
        print "    arg: %s" % arg

def f_param_nome_variaveis( p1, **args ):
    print "p1: '%s'" % p1
    for p_name, p_value in args:
        print "    arg: %s=%s" % ( p_name, p_value )
```

Classes:

```
class A:
    attributo = 1
    __privado = 123
    def __init__( self, valor ):
        self.attributo = valor
        self.__metodo_privado()
    def __metodo_privado( self ):
        print "chamando metodo privado"
class B:
    attributo = 2
    def __init__( self ):
        self.novo_attributo = 2
class C( A, B ):
    def __init__( self ):
        B.__init__( self )
class D( B, A ):
    def __init__( self ):
        B.__init__( self )
a = A( 1 )
b = B()
c = C()
d = D()
print a.attributo   # imprime 1
print b.attributo   # imprime 2
print c.attributo   # imprime 1 <- herança múltipla (A,B) A sobrepõe-se a B
print d.attributo   # imprime 2 <- herança múltipla (B,A) B sobrepõe-se a A
```

Módulos e Espaço de Nomes:

```
import urllib
url      = "http://www.unicamp.br/" + urllib.quote( "index.html" )
conteudo = urllib.urlopen( url ).read()

# importa símbolos para espaço de nomes atual:
from urllib import *
url      = "http://www.unicamp.br/" + quote( "index.html" )
conteudo = urlopen( url ).read()
```