



Curso de Python

25/03/10

Rudson R. Alves
Mestre em Física

Prof. na UVV
<http://rra.etc.br>

Introdução

Esta apresentação é baseada no material:

- ♦ *Python na Prática: Um curso objetivo de programação em Python* de Christian Robottom Reis:
<http://www.async.com.br/projects/python/pnp/>
- ♦ *Tutorial Python* de Guido van Rossum, criador do Python

Cronograma

- O que é *Python*
- Por que *Python*
- *Python* Básico
- Estruturas de Controle
- Exceções
- Funções
- Escopo de Variáveis
- Funções Pré-definidas
- *Docstrings*
- Manipulação de Arquivos
- Orientação a Objetos
- Importando Módulos

O que é Python: Linguagem interpretada

Classificação das linguagens:

- compilada
- interpretada

Compiladas:

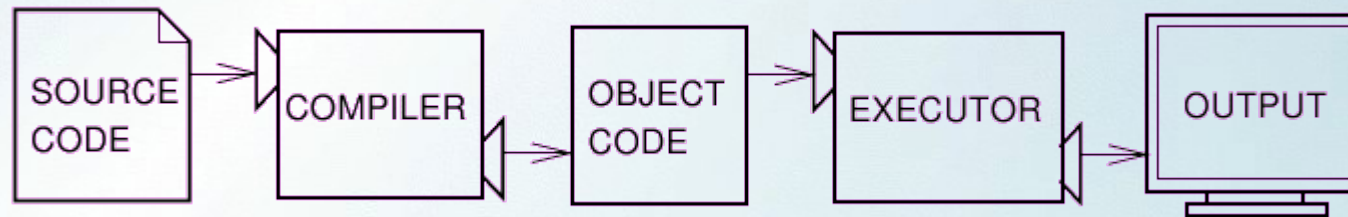
Fortran, C, C++, Visual Basic, ...

```
#include <iostream>

using namespace std;

int main ()
{
    cout << "Hello World! \n";
    return 0;
}
```

```
$ gcc hello.c -o hello
$ ./hello
Hello World
```



O que é Python: Linguagem interpretada

Interpretadas:

Python, Perl, Basic tradicional, Shell Script, ...

```
$ python hello.py  
Hello World
```



O que é Python: Tipagem dinâmica

Python possui o que se chama de **tipagem dinâmica**, ou seja, a tipagem pode mudar a cada nova entrada de dados em uma variável.

```
>>> a = 1
>>> type(a)
<type 'int'>
>>> a = 'abacaxi'
>>> type(a)
<type 'str'>
>>> a = 1.0
>>> type(a)
<type 'float'>
```

A tipagem dinâmica reduz a quantidade de tempo de planejamento prévio e é um mecanismo importante para garantir flexibilidade e simplicidade das funções *Python*.

O que é Python: Delimitação por indentação

Em *Python* não existe um delimitador específico para blocos de código. A delimitação é feita pela indentação:

```
print "O valor de a é "  
if a == 0:  
    print "zero"  
    a = "zero"  
else:  
    print a  
  
exit()
```

Isto garante que o código seja sempre legível.

O que é Python: Tipos de alto nível

Além dos tipos básicos (inteiro, ponto flutuante, ...), o *Python* possui ainda outros tipos de mais alto nível:

Listas []: como um vetor em outras linguagens, lista é um conjunto de valores acessados por um índice numérico, inteiro, iniciado por zero. Em *Python*, uma lista ainda podem armazenar todo tipo de valores.

```
>>> a = ["A", "B", "C", 1, 2, 3, 4.0]
>>> print a[0]
A
>>> a[6]
4.0
>>> a
['A', 'B', 'C', 1, 2, 3, 4.0]
>>> type(a[6])
<type 'float'>
```


O que é Python: Tipos de alto nível

Tuplas: *Tuplas* são seqüências de elementos arbitrários como listas, com a exceção de que são **imutáveis**.

Strings: *string* em *Python* é uma seqüência imutável, alocada dinamicamente e sem restrição de tamanho.

Dicionários: dicionários são seqüências que podem utilizar índices (imutáveis) de tipos variados. conhecidos como *arrays* associativos.

Arquivo: *Python* possui um tipo pré-definido para manipular arquivos. Este tipo permite que o arquivo seja facilmente lido, alterado e escrito.

Classess e Instâncias: classes são estruturas especiais que servem para apoiar programação orientada a objetos. Instâncias são expressões concretas destas classes.

O que é Python: Orientação a Objetos

Em *Python*, todos os dados podem ser considerados objetos. Por exemplo, toda *string* possui o método *upper*:

```
>>> a = 'laranja'
>>> a.upper()
'LARANJA'
>>> 'camaleao'.upper()
'CAMALEAO'
```

Da mesma forma inteiros, ponto flutuante, *tuplas*, dicionários, listas, ..., são todos objetos. O comando `dir(variável)` mostra os métodos disponíveis.

```
>>> dir("15") # métodos da string constante "15"
['__add__', '__class__', '__contains__', '__delattr__', '__doc__',
 '__eq__', '__ge__', '__getattribute__', '__getitem__',
 '__getnewargs__', '__getslice__', '__gt__', '__hash__', '__init__',
 '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
 '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__',
 '__rmul__', '__setattr__', '__str__', 'capitalize', 'center', 'count',
 'decode', 'encode', 'endswith', 'expandtabs', 'find', 'index',
 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle',
 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace',
 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate',
 'upper', 'zfill']
```


Por que Python

Como existe muitas linguagens diferentes, por que aprender *Python*?

- Os conceitos fundamentais da linguagem são simples de entender;
- A sintaxe do *Python* é clara e fácil de aprender;
- Os tipos pré-definidos em *Python* são poderosos e simples de usar;
- O interpretador *Python* permite aprender e testar rapidamente trechos de código
- *Python* é expressivo, com abstrações de alto nível => código curto e rápido desenvolvimento;

Por que Python

- Existe suporte para uma diversidade grande de bibliotecas (*Qt*, *GTK*, *web*, *db*, ...);
- É fácil escrever extensões para *Python* em *C* e *C++*, quando for necessário desempenho máximo, ou quando necessitar de interfacear alguma ferramenta nestas linguagens;
- *Python* permite que o programa execute em múltiplas plataformas, sem alterações;
- Possui tratamento de exceções (moderno mecanismo de tratamento de erros);
- *Python* é orientado a objetos (incluindo herança múltiplas).
- *Python* é livre.

Python Básico

Nesta seção será abordado aspectos essenciais da linguagem, como tipos, operadores e estruturas.

Comentários em *Python* seguem a mesma estrutura dos comentários em *bash script*:

```
>>> # isto    um comentário
```

Python Básico: O interpretador Python

O *Python* permite executar comandos diretamente através de seu interpretador, ou uma lista de comandos, armazenada em um arquivo (programa em *Python*)

Para chamar o interpretador *Python* apenas digite “python” no *prompt* do *shell*:

```
$ python
Python 2.5.1 (r251:54863, Sep  4 2007, 19:00:19)
[GCC 4.1.2] on linux2
Type "help", "copyright", "credits" or "license" for
more information.
>>>
```

A mensagem que irá aparecer deve ser bem parecida com a mensagem acima, a menos das versões do *Python* e do *GCC*.

“>>>” e “...” são os *prompts* do *Python*.

Python Básico: O interpretador Python

Mantendo a tradição vamos fazer o “*Hello World*”

```
>>> a = "Hello"  
>>> b = "World"  
>>> print a, b  
Hello World
```

Observe que o comando *print* insere um espaço automaticamente entre as duas variáveis. O mesmo poderia ter sido feito com os comandos abaixo:

```
>>> a = "Hello World"  
>>> print a  
Hello World  
>>> print "Hello World"  
Hello World  
>>> "Hello World"  
'Hello World'
```

Python Básico: Criando um programa Python

O mesmo pode ser feito através da criação de um módulo (como são chamados os programas em *Python*). Em um editor de sua escolha escreva:

```
# Primeiro programa Python: hello.py  
a = "Hello"  
b = "World"  
print a, b
```

Salve o programa com o nome *hello.py* e execute-o chamando o interpretador:

```
$ python hello.py  
Hello World  
$ _
```

Python Básico: Criando um programa Python

O interpretador pode ser chamado automaticamente pelo sistema. Para isto acrescente o *path* dele no início programa e o torne executável:

```
#!/usr/bin/python
# Primeiro programa Python: hello.py
a = "Hello"
b = "World"
print a, b
```

```
$ chmod +x hello.py
$ ./hello.py
Hello World
$ _
```

Se tiver dúvidas quando a localização do interpretador *Python*, use o comando: *which python*

Python Básico: Criando um programa Python

Uma versão em português, pode trazer algumas surpresas:

```
#!/usr/bin/python
# Primeiro programa Python: ola.py
a = "Olá"
b = "Mundo"
print a, b
```

```
$ ./ola.py
  File "./ola.py", line 3
SyntaxError: Non-ASCII character '\xe1' in file
./ola.py on line 3, but no encoding declared; see
http://www.python.org/peps/pep-0263.html for details
$ _
```

Isto ocorre por conta do acento no 'Olá', passado à variável **a**. Para adicionar suporte aos caracteres acentuados, adicione a linha abaixo ao início do seu programa:

```
#-*- coding: iso-8859-1 -*-
```

Python Básico: Criando um programa Python

O programa deve ficar assim:

```
#!/usr/bin/python
#-*- coding: iso-8859-1 -*-
# Primeiro programa Python: ola.py
a = "Olá"
b = "Mundo"
print a, b
```

```
$ ./ola.py
Olá Mundo
$ _
```

Python Básico: Tipos numéricos

O *Python* possui alguns tipos numéricos pré-definidos: inteiros (*int*), ponto flutuante (*float*), booleanos (*bool*) e complexos (*complex*). Este tipos suportam as operações matemáticas básicas.

```
>>> a, b = 1, 2.5      # atribui 1 a "a" e 2.5 a "b"
>>>                    # um inteiro e um ponto flutuante
>>> c = True           # booleano
>>> z = 3 + 4j          # complexo
>>>
>>> a + b              # resultado em ponto flutuante
3.5
>>> int(a + b)         # resultado inteiro
3
>>> b * z              # resultado complexo
(7.5+10j)
>>> type(z)            # mostra a tipagem da variável
<type 'complex'>
```


Python Básico: Tipos numéricos

Python também trabalha com base octal (0##) e hexadecimal (0x##)

```
>>> a = 010
>>> a
8
>>> 0xFF
255
```

Um número real deve possuir um ponto “.”

```
>>> 5/2                # resultado inteiro
2
>>> 5/2.               # resultado ponto flutuante
2.5
>>> 5*2.13             # resultado ponto flutuante
10.649999999999999
```

Especial atenção aos números como 2.13, cujo a representação decimal no computadores é limitada.

Python Básico: Tipos numéricos

Python como uma calculadora e mais operações:

```
>>> 25/3.          # resultado ponto flutuante
8.333333333333333
>>> _+2            # adiciona 2. ao resultado anterior
10.333333333333334
>>> type(_)        # verifica o tipo do resultado ant.
<type 'float'>
```

Tipagem dinâmica:

```
>>> a = 3
>>> type(a)
<type 'int'>
>>> a = 3.1415
>>> type(a)
<type 'float'>
>>> a = "isto e' uma string"
>>> type(a)
<type 'str'>
```

```
>>> a = 3 + 4j
>>> type(a)
<type 'complex'>
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a)
5.0
```

Python Básico: Listas []

Lista é uma seqüência de valores indexadas por um inteiro. Uma lista pode conter qualquer tipo de valor, incluindo valores de tipos mistos:

```
>>> numeros = [1, 2, 3]
>>> nomes = [ 'alberto', 'carlos', 'simone' ]
>>> misto = [ 1,2,3.0,'alberto','carlos',3.5,'simone' ]
>>> listas = [ numeros, nomes, misto ]
>>> print listas
[[1, 2, 3], ['alberto', 'carlos', 'simone'], [1, 2, 3.0, 'alberto', 'carlos', 3.5, 'simone']]
```

Os elementos da lista podem ser acessados por meio de índices que vão de 0 até o comprimento da lista-1:

```
>>> len(numeros)-1
2
>>> numeros[0]
1
>>> numeros[2]
3
```

```
>>> listas[0]
[1, 2, 3]
>>> listas[1]
['alberto', 'carlos', 'simone']
>>> listas[1][1]
'carlos'
```


Python Básico: Listas [] - seleções

Python também permite acessar uma lista de trás para frente, identificando o índice por um sinal de menos:

```
>>> numeros[-1]      # último valor
3
>>> numeros[-2]      # o penúltimo valor
2
```

Fatias, ou *slices*, de uma lista podem ser geradas facilmente com o “:”

```
>>> n = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 ]
>>> n[2:4]
[3, 4]
>>> n[6:]
[ 7, 8, 9, 0]
>>> n[-5:-1]
[6, 7, 8, 9]
```

Python Básico: Listas [] - métodos

Como foi dito anteriormente, em *Python* tudo é um objeto. Vamos ver alguns métodos de uma lista:

```
>>> numeros.append(0)
>>> numeros
[1, 2, 3, 0]
>>> numeros.sort()
>>> numeros
[0, 1, 2, 3]
>>>
```

```
>>> numeros.reverse()
>>> numeros
[3, 2, 1, 0]
>>> numeros.pop()
0
>>> numeros
[3, 2, 1]
```

Para conhecer todos os métodos de um objeto utilize o comando *dir(objeto)*

```
>>> dir(numeros)
['__add__', '__class__', '__contains__', '__delattr__',
 '__delitem__', '__delslice__', '__doc__', '__eq__', '__ge__',
 '__getattr__', '__getitem__', '__getslice__', '__gt__',
 '__hash__', '__iadd__', ..., 'append', 'count', 'extend',
 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

Python Básico: Listas [] - métodos

Outros métodos de listas:

```
>>> n = [ 1, 2, 3]
>>> m = [ 4, 5, 6]
>>> n.extend(m); n
[1, 2, 3, 4, 5, 6]
>>> n.insert(2, 'galo'); n
[1, 2, 'galo', 3, 4, 5, 6]
>>> n.remove(4); n
[1, 2, 'galo', 3, 5, 6]
>>> n.append('barco'); n
[1, 2, 'galo', 3, 5, 6, 'barco']
>>> n.pop(); n.pop(2)
'barco'
'galo'
>>> n.index(5)
3
```

```
>>> n.append(5)
>>> n.extend(m)
>>> n
[1, 2, 3, 5, 5, 4, 5, 6]
>>> n.count(5)
3
>>> n.sort(); n
[1, 2, 3, 4, 5, 5, 5, 6]
>>> n.reverse(); n
[6, 5, 5, 5, 4, 3, 2, 1]
```


Python Básico: Tuplas ()

Tuplas são como listas, um conjunto de elementos acessíveis por um índice inteiro.

Lista

```
>>> n = [1,2,3]
>>> n
[1, 2, 3]
>>> n[0]='laranja'; n
['laranja', 2, 3]
```

Tupla

```
>>> t = (1,2,3)
>>> t
(1, 2, 3)
>>> t[0]
1
```

No entanto *tuplas* são imutáveis

```
>>> t[0]='laranja'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item
assignment
```

Python Básico: Strings

String é uma sequência imutável com o propósito especial de armazenar cadeias de caracteres.

```
>>> a = 'abcdefghij'
>>> a[0]
'a'
>>> a[-1]
'j'
>>> a[6:]
'ghij'
```

```
>>> len(a)
10
>>> a.__len__()
10
>>> a.upper()
'ABCDEFGHIJ'
```

Você não consegue alterar um elemento da *string*

```
>>> a[1]='B'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item
assignment
```

Python Básico: Strings - algumas operações

Colar strings (+)

```
>>> b = 'klmnopqrst'
>>> a + b
'abcdefghijklmnopqrst'
>>> print a, b
abcdefghijklmnopqrst
>>> c = (a + b).upper()
>>> print c
ABCDEFGHIJKLMNOPQRST
```


Python Básico: Dicionários {}

Um dicionário é uma coleção de elementos, onde é possível utilizar um índice de qualquer tipo imutável.

```
>>> telefones = { "pedro" : 33212121, "patricia" :  
34000022, "fernanda" : 88222298 }  
>>> print telefones["fernanda"]  
88222298
```

Alguns métodos de dicionários:

```
>>> telefones.keys()  
['pedro', 'patricia', 'fernanda']  
>>> telefones.values()  
[33212121, 34000022, 88222298]  
>>> telefones.items()  
[('pedro', 33212121), ('patricia', 34000022),  
( 'fernanda', 88222298)]  
>>> telefones.has_key('alberto')  
False
```

Python Básico: Operadores - aritméticos

```
>>> 7 + 3          # adição
10
>>> 7 - 3          # subtração
4
>>> 8 % 3          # resto da divisão
2
>>> 8 / 3          # divisão inteira
2
>>> 8 / 3.         # divisão em ponto flutuante
2.6666666666666665
>>> 8 * 3          # produto
24
>>> 8 ** 2         # exponenciação
64
```

Python Básico: Operadores - aritméticos

Strings

```
>>> a = 'bits'
>>> a * 2
'bitsbits'
>>> print '64 ' + a
64 bits
```

Listas

```
>>> a = [5, 6, 7, 8]
>>> b = [9, 10]
>>> print b * 2
[9, 10, 9, 10]
>>> print a + b
[5, 6, 7, 8, 9, 10]
```

Tuplas

```
>>> a = (2, 3, 4)
>>> print a + (5, 6)
(2, 3, 4, 5, 6)
```


Python Básico: Operadores - bits

Cadeias de *bits*

```
>>> a = 0x1F
>>> b = 0x01
>>> print a, b
31, 1
>>> a & b          # e
1
>>> a | b          # ou
31
>>> b << 4         # 4 deslocamentos para a esquerda
16
>>> a >> 4         # 4 deslocamentos para a direita
1
>>> ~a             # inversão em complemento de 2
-32
```

Python Básico: Atribuição e condicionais

```
>>> a = 1
>>> a += 1
>>> print a
2
>>> a *= 10
>>> print a
20
>>> a /= 2
>>> print a
10
>>> a, b = 3, 5
>>> a, b = b, a+b
>>> print a, b
5 8
```

A última atribuição é o mesmo que fazer:

```
>>> b = a+b
>>> a = b
```

Atribuição condicionada -
Python 2.5

```
>>> a, b = 5, 3
>>> c = a if a > b else b
>>> print c
5
```

Condicionais: *Booleano*

```
>>> 2 == 4
False
>>> 2 != 4
True
>>> 2 > 4
False
>>> 2 < 4
True
>>> 3 <= a
True
```

Python Básico: Combinação de operadores

```
>>> a, b = 5, 3; a, b
(5, 3)
>>> 0 < a < b           # avaliação é feita da esquerda
False
>>> 0 < a > b           # para a direita
True
```

Mais operadores...

```
>>> nome = 'pedro'
>>> idade = 25
>>> nome == 'pedro' and idade == 25
True
>>> len(nome) < 10 and idade > 30
False
>>> len(nome) < 10 or idade > 30
True
```


Python Básico: Combinação de operadores

Mais uma condicional.....

```
( 'string_1', 'string_2' ) [ condição ]
```

Se a condição for:

- ▶ verdadeira: 'string_2'
- ▶ falsa: 'string_1'

```
>>> pedro_age = 15
>>> jose_age = 20
>>> ("Pedro", "Jose") [ pedro_age > jose_age ]
'Pedro'
>>> pedro_age = 25
>>> ("Pedro", "Jose") [ pedro_age > jose_age ]
'Jose'
```

```
# em um programa pygtk – para testar um ToggleButton:
print "%s was toggled %s" % (data, ("OFF", "ON")[widget.get_active()])
```

Python Básico: Operadores - %

Substituição em *strings*: operador %

```
>>> print "\nNome: %s" % "Roberta"
```

```
Nome: Roberta
```

```
>>> a = "Total de itens: %d"
```

```
>>> b = "Custo: %5.2f"
```

```
>>> print a % 10; print b % 25.83
```

```
Total de itens: 10
```

```
Custo: 23.83
```

```
>>> print "\nCliente: %s, Valor: %5.2f" % ("Alberto",  
25.45)
```

```
Cliente: Alberto, Valor: 25.45
```

Estruturas de Controle: if

A estrutura condicional *if* usa a sintaxe abaixo:

```
if condição:
    # comandos
    ...
elif condição:
    # comandos
    ...
else:
    # comandos
    ...
```

Observe que quem delimita o bloco é a indentação.

```
>>> a = 5; b = 8
>>> if a > b:
...     print "a é maior que b"
...     c = "maior"
... elif a == b:
...     print "a é igual a b"
...     c = "igual"
... else:
...     print "a é menor que b"
...     c = "menor"
...
a < b
>>> print a,c,b
5 menor 8
```


Estruturas de Controle: for

O laço *for* do *Python* é semelhante ao *for* do *bash*. Ele percorre uma seqüência de elementos:

```
for variável in seqüência:  
    # comandos  
    ...
```

```
>>> lista = [ "pedra", 12, 54.56, 3 + 5j ]  
>>> for item in lista:  
...     print item  
pedra  
12  
54.56  
(3+5j)  
  
>>> for i in range(1,5):  
...     print i,  
1 2 3 4
```

Estruturas de Controle: for

```
>>> dict = {"batata": 500, "abóbora": 1200, "cebola": 800}
>>> for e in dict.keys():
...     print "Item: %8s  Peso: %8s" % (e, dict[e])
...
Item:   batata  Peso:      500
Item: abóbora  Peso:     1200
Item:   cebola  Peso:      800
```

```
>>> animais = ["gato", "jaguaririca", "salamandra", "ran"]
>>> for a in animais:
...     print "%12s: %3d" % (a, len(a))
...
      gato:    4
jaguaririca:  11
salamandra:  10
      ran:     3
```

Estruturas de Controle: for

O laço for ainda aceita as instruções *continue* e *break*. Sua sintaxe completa tem a forma:

```
for variável in seqüência:
    # bloco de comandos
    ...
else:
    # bloco de comandos na ausência de um break
    ...
```

```
>>> var = [2, 4, 5, 6, 7, -3, 4, 8, 3]
>>> for v in var:
...     if v < 0:
...         print "Valor negativo encontrado: %d" % v
...         break
...     else:
...         print "Nenhum negativo encontrado"
...
Valor negativo encontrado: -3
```


Estruturas de Controle: for

Números primos menores que 30:

```
>>> for n in range(2, 30):  
...     for x in range(2, n):  
...         if n % x == 0:  
...             break  
...     else:  
...         # loop fell through without finding a factor  
...         print "%2d é um número primo" % n  
...  
2 é um número primo  
3 é um número primo  
5 é um número primo  
7 é um número primo  
11 é um número primo  
13 é um número primo  
17 é um número primo  
19 é um número primo  
23 é um número primo  
29 é um número primo
```

Estruturas de Controle: while

O laço *while* é útil quando se é necessário fazer um teste a cada interação do laço. Assim como o *for*, aceita as instruções *continue* e *break*. Sua sintaxe completa tem a forma:

```
while condição:
    # bloco de comandos
    ...
else:
    # bloco de comandos executados na ausência de um break
    ...
```

```
>>> m = 3 * 19; n = 5 * 13
>>> count = 0
>>> while m < n:
...     m = n / 0.5
...     n = m / 0.5
...     count += 1
...
>>> print "Foram %d inerações" % count
Foram 510 inerações
```

Estruturas de Controle: while

Série de Fibonacci até 20:

```
>>> a, b, n = 0, 1, 20      # o mesmo que: a = 0; b = 1; c = 20
>>> fib=""
>>> while b < n:
...     fib = fib + " + " + str(b)
...     a, b = b, a+b      # a = b; b = a + b
...
>>> print "A série de Fibonacci até %d, é:\n%s" % ( n, fib)
A série de Fibonacci até 20, é:
1 1 2 3 5 8 13
```


Exceções

Com os laços *for* e *while*, e as condicionais *ifs*, todas as necessidades de controle em um programa podem ser implementadas. Mas quando algo inesperado ocorre, *Python* oferece uma forma adicional de controlar o fluxo de execução: a **exceção**

```
>>> a = [1, 2, 3]
>>> print a[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

A primeira linha anuncia que ocorreu um ***traceback***. A segunda linha indica a linha de código e o arquivo onde o erro ocorreu (*stdin* – entrada padrão, modo interativo). Na terceira linha indica o tipo de exceção levantada - ***IndexError***.

Exceções: Tratando exceções

A sintaxe para tratamento de exceções é apresentada abaixo:

```
try:
    # comandos que podem gerar a exceção
    ...
except tipo_exceção [, variável]:
    # comandos a serem executados para a exceção gerada
    ...
```

A linha *except* também pode conter uma *tupla* com diversas exceções:

```
except (tipo_exceção_1, tipo_exceção_2, ...) [, variável]:
    # comandos a serem executados para a exceção gerada
    ...
```

Exceções: Tratando exceções

A exceção anterior poderia ser tratada da forma:

```
>>> a = [1, 2, 3]
>>> try:
...     print a[5]
... except IndexError:
...     print "Tentativa de acessar um índice inexistente."
...
Tentativa de acessar um índice inexistente.
```

Outro exemplo:

```
>>> a = "tolo"
>>> print a + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

```
>>> try:
...     print a + 1
... except TypeError:
...     print "Não pode somar uma string a um inteiro."
...
Não pode somar uma string a um inteiro.
```


Funções

Sintaxe geral de uma função:

```
def nome_função(arg_1, arg_2, ..., arg_n):  
    # código da função  
    ...  
    [return valor_de_retorno]
```

O retorno é opcional. Uma função sem retorno pode ser encarada como um procedimento.

```
>>> def fibonacci(n):  
...     a, b = 0, 1  
...     while b < n:  
...         print b,  
...         a, b = b, a+b  
...  
>>> fibonacci(100)  
1 1 2 3 5 8 13 21 34 55 89
```

Funções

```
>>> def imprime_cardapio(pratos):  
...     "Procedimento para impressao do cardapio"  
...     print "Cardapio do dia\n"  
...     for p in pratos:  
...         imprime_prato(p)  
...     print "\nTotal de pratos: %d" % len(pratos)  
...  
>>> def imprime_prato(p):  
...     "Procedimento para impressao do prato"  
...     print "%20s ..... %6.2f" % (p["nome"], p["preco"])  
>>>  
>>> p1 = {"nome" : "Arroz com brocolis", "preco" : 9.90}  
>>> p2 = {"nome" : "Sopa de legumes", "preco" : 8.70}  
>>> p3 = {"nome" : "Lentilhas", "preco" : 7.80}  
>>>  
>>> lista_pratos = [p1, p2, p3]
```

Observe que a primeira linha, em cada função (procedimento) é apenas descritiva, não sendo executada pelo interpretador. Esta linha é chamada de *docstring*, uma documentação para a função.

Funções

```
>>> imprime_cardapio(lista_pratos)
Cardapio do dia

Arroz com brocolis ..... 9.90
  Sopa de legumes ..... 8.70
    Lentilhas ..... 7.80

Total de pratos: 3
>>>
>>> imprime_cardapio.__doc__
'Procedimento para impressao do cardapio'
>>> imprime_prato.__doc__
'Procedimento para impressao do prato'
```

Fibonacci com retorno

```
>>> def fibonacci(n):
...     a, b, f = 0, 1, ""
...     while b < n:
...         a, b, f = b, a+b, f+str(b)+" "
...     return f
```


Funções: argumento com valor padrão

É possível definir valores padrões para os argumentos de entrada:

```
>>> def aplicar_multa(valor, taxa=0.15):  
...     return valor*(1. + taxa)  
...  
>>> print "Valor a pagar: %5.2f" % aplicar_multa(100)  
115.00  
>>> print "Valor a pagar: %5.2f" % aplicar_multa(100, .25)  
125.00
```

Não utilize como valor padrão listas, dicionários ou outros valores mutáveis. O resultado alcançado pode não ser o desejado.

Funções: argumento com valor padrão

Veja outro exemplo:

```
>>> from math import sqrt
>>> def segrau(a, b = .0, c = .0):
...     delta = b**2 - 4*a*c
...     if delta > 0:
...         r1 = (-b + sqrt(delta))/(2*a)
...         r2 = (-b - sqrt(delta))/(2*a)
...     else:
...         r1 = complex(-b/(2*a), sqrt(-delta)/(2*a))
...         r2 = complex(-b/(2*a), -sqrt(-delta)/(2*a))
...     return ( r1, r2 )
...
```

```
>>> segrau(2)
(0.0, -0.0)
>>> segrau(2, 4)
(0.0, -2.0)
>>> segrau(2, 5, 3)
(-1.0, -1.5)
>>> segrau(2, c = 2)
(1j, -1j)
```

```
>>> segrau(a=2)
(0.0, -0.0)
>>> segrau(b=4, a=2)
(0.0, -2.0)
>>> segrau(c=3, a=2, b=5)
(-1.0, -1.5)
>>> segrau(c=2, a=2)
(1j, -1j)
```

Funções: argumento com valor padrão

Esta função também poderia ser chamada nas formas:

```
>>> segrau(a=2)
(0.0, -0.0)
>>> segrau(b=4, a=2)
(0.0, -2.0)
>>> segrau(c=3, a=2, b=5)
(-1.0, -1.5)
>>> segrau(c=2, a=2)
(1j, -1j)
```

As variáveis “b” e “c” possuem valor padrão e por isto são opcionais, já a variável “a” é obrigatória e deve ser passada:

```
>>> segrau(a=4)
(0.0, -0.0)
>>> segrau(b=2, c=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: segrau() takes at least 1 non-keyword argument (0
given)
```


Funções: conjunto de argumentos

Um conjunto de argumentos opcionais podem ser passados com o auxílio do curinga “*”:

```
>>> def media(*valores):  
...     soma = 0.  
...     for n in valores:  
...         soma += n  
...     return soma/len(valores)  
...  
>>> media(1,2,3,4,5,6)  
3.5
```

Funções: dois conjuntos de argumentos

Um segundo conjunto de argumentos opcionais podem ser passados usando “**”, para o segundo conjunto. Pelo que pode perceber, este segundo conjunto deve ser um composto de:

`var_0='valor_0', var_1='valor_1', ...`

Se for usar “*” e “**”, tome o cuidado para que “*” sempre preceda “**”.

```
>>> def teste1(nome, **numeros):
...     print nome + ":"
...     for i in numeros.keys():
...         print '%10s : %d' % (i, numeros[i])
...
>>> def teste2(nome, *strings, **numeros):
...     print nome + ":"
...     for i in strings:
...         print '> ', i
...     keys = numeros.keys()
...     keys.sort()
...     for i in keys:
...         print '%10s : %d' % (i, numeros[i])
...
```

Funções: dois conjuntos de argumentos

```
>>> teste1('Numeros', um=1, dois=2, tres=3, quatro=4, cinco=5)
Numeros:
```

```
    um : 1
cinco : 5
  tres : 3
quatro : 4
    dois : 2
```

```
>>>
```

```
>>> teste2('Numeros', 'Os números são ordenados', 'pelos seus
nomes', um=1, dois=2, tres=3, quatro=4, cinco=5)
```

```
Numeros:
```

```
> Os números são ordenados
```

```
> pelos seus nomes
```

```
    cinco : 5
      dois : 2
quatro : 4
  tres : 3
    um : 1
```

```
>>>
```

```
>>> teste2('Numeros', um=1, dois=2, tres=3, quatro=4, cinco=5)
```


Escopo da Variável

Quanto uma variável é definida no bloco principal de um programa, ele estará presente no escopo de todas as funções definidas a posteriori:

Inicie uma nova seção do Python
para as discussões a seguir

```
>>> a = 5
>>> b = 8
>>> def soma(x,y):
...     print 'a =',a, 'b =', b
...     return x+y
...
>>> soma(2, 7)
a = 5 b = 8
9
```

Observe que os valores das variáveis **a** e **b** são impressas corretamente, mesmo elas não tendo sido passadas para a função, ou seja, estas variáveis fazem parte do escopo da função **soma**.

Escopo da Variável

Continuando o exemplo, observe que ao atribuirmos valores a **a** e **b**, de dentro da função **produto**, os valores não são sobre-escritos aos valores das variáveis de mesmo nome, no escopo principal:

```
>>> def produto(x,y):  
...     a, b = x, y  
...     print 'produto: a =',a, 'b =', b  
...     return a * b  
...  
>>> produto(2, 7)  
produto: a = 2 b = 7  
14  
>>> print 'a =',a, 'b =', b  
a = 5 b = 8
```

Escopo da Variável

A função `globals()` retorna um dicionário com todo o escopo global e os seus valores.

```
>>> globals()
{'a': 5, 'b': 8, '__builtins__': <module '__builtin__' (builtin)>, 'produto': <function produto at 0x2b620fa4be60>, 'soma': <function soma at 0x2b620fa4bb18>, '__name__': '__main__', '__doc__': None}
```

A função `locals()` retorna um dicionário semelhante ao da função `globals()`, mas para escopo local.

```
>>> def divisao(x,y):
...     a, b = x, y
...     print a/b
...     return locals()
...
>>> divisao(10,3)
3
{'a': 10, 'y': 3, 'b': 3, 'x': 10}
>>> print 'a =',a, 'b =', b
a = 5 b = 8
```


Escopo da Variável

A função `global()` permite definir uma variável no escopo global, de dentro de uma função:

```
>>> def pot(x,n):  
...     global p  
...     p = n  
...     return x**p, locals()  
...  
>>> pot(3,3)  
(27, {'x': 3, 'n': 3})  
>>> print p  
3  
>>> print n  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
NameError: name 'n' is not defined
```

Escopo da Variável

Vamos testar a função `locals()` na função `teste3`. Observe que `**numeros` é passado como um dicionário e `*strings` como uma tupla:

```
>>> def teste2(nome, *strings, **numeros):
...     print nome + ":"
...     for i in strings:
...         print '> ', i
...     keys = numeros.keys()
...     keys.sort()
...     for i in keys:
...         print '%10s : %d' % (i, numeros[i])
...     return locals()
...
>>> teste2('Numeros', 'Os números são ordenados', 'pelos seus
nomes', um=1, dois=2, tres=3, quatro=4, cinco=5)
Numeros:
> Os números são or...
{'keys': ['cinco', 'dois', 'quatro', 'tres', 'um'], 'i': 'um',
'numeros': {'um': 1, 'cinco': 5, 'tres': 3, 'quatro': 4,
'dois': 2}, 'strings': ('Os números são ordenados', 'pelos
seus nomes'), 'nome': 'Numeros'}
```

Funções Pré-definidas

O Python possui várias funções pré-definidas, que não necessitam de importações externas. Vou passar rapidamente algumas destas funções:

- **range(a[, b[, c]]):** retorna uma lista de inteiros de 0 a a-1, caso somente a seja passado como argumento; de a até b, caso a e b sejam passados como argumentos; a até b com o incremento c, caso a, b e c sejam passados como argumentos.

```
>>> range(10)          # gera uma lista com elementos de 0 a 9
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
>>> range(3, 10)       # gera uma lista de 3 a 9
[3, 6, 9, 12, 15, 18, 21, 24, 27]
>>>
>>> range(30,3, -3)    # gera uma lista de 30 a 4 com step -3
[30, 27, 24, 21, 18, 15, 12, 9, 6]
```


Funções Pré-definidas

- **xrange(a[, b[, c]])**: o mesmo que range, mas ao invés de retornar uma lista, retorna um objeto que gera os números desejados por demanda. Portanto não serão gerados os elementos de uma lista. A grande utilidade do xrange é a sua aplicação em loops. Com o xrange, um loop pode ficar até 25% mais rápido.

Considere os dois loops de 1.000.000 interações abaixo:

```
#!/usr/bin/python
# Loop range.py
for i in range(1000000):
    pass
```

```
#!/usr/bin/python
# Loop xrange.py
for i in xrange(1000000):
    pass
```

```
$ time range.py | grep user
user      0m0.457s

$ time xrange.py | grep user
user      0m0.337s
```

Funções Pré-definidas

- **len(a)**: retorna o comprimento da variável a. Se a for uma lista, tupla ou dicionário, len retorna o seu número de elementos.

```
>>> a = 'Alberto Santos Dumont'
>>> len(a)
21
>>> a = [1,2,3]
>>> len(a)
3
```

- **round(a[, n])**: arredonda o real 'a' com 'n' casas decimais. Se 'n' for omitido, será considerado n = 0.

```
>>> round(5.48)
5.0
>>> round(5.548)
6.0
>>> round(5.548, 1)
5.5
```

Funções Pré-definidas

- **pow(a, n):** retorna o valor de a^n , onde **a** e **n** podem ser inteiro, real ou complexo. O mesmo que $a^{**}n$.

```
>>> pow(2,3)
8
>>> pow(2.0,3.3)
9.8491553067593287
>>> pow(2.0,3.3j)
(-0.65681670994609054+0.75405026989955604j)
```

- **chr(a):** retorna o caracter ascii correspondente ao código **a**. O valor de **a** deve ser um inteiro entre 0 e 255.

```
>>> for i in range(10): print chr(65+i),
...
A B C D E F G H I J
```


Funções Pré-definidas

- **unichr(a)**: como a função anterior, retorna o caractere Unicode correspondente ao inteiro **a**. O valor de **a** deve estar entre 0 e 65535. Até 127, `chr(a) == unichr(a)`

```
>>> for i in range(10): print unichr(65+i),  
...  
A B C D E F G H I J
```

- **ord(a)**: retorna o código ascii do caracter passado pela variável **a**. O valor de **a** deve ser apenas um caracter.

```
>>> for i in range(10): a = chr(i+65); print '%s : %d' % (a,  
ord(a))  
...  
A : 65  
B : 66  
C : 67  
D : 68  
E : 69  
F : 70 ...
```

Funções Pré-definidas

- **min(a, b)**: retorna o menor valor entre **a** e **b**. Funciona para qualquer tipo de variáveis. No caso de comparação entre tipos diferentes, a comparação é feita após converter os argumentos em cadeias de caracteres.

```
>>> min(1,6)
1
>>> min('a', 2)
2
>>> min('abacate', 'flores')
'abacate'
```

- **max(a, b)**: retorna o maior valor entre **a** e **b**. Funciona de forma a função min acima.

```
>>> max(1,6)
6
>>> max('a', 2)
'a'
>>> max('abacate', 'flores')
'flores'
```

Funções Pré-definidas

- **abs(a)**: retorna o valor absoluto de **a**, seu módulo. Esta função somente trabalha com números inteiros, reais e complexos.

```
>>> abs(-3)
3
>>> abs(-3.0)
3.0
>>> abs(3-4j)           # módulo de um complexo
5.0
```

- **hex(a)** e **oct(n)**: retorna o valor hexadecimal e octal da variável **a**.

```
>>> hex(22)
'0x16'
>>> oct(22)
'026'
>>> hex(022)
'0x12'
```


Funções Pré-definidas: conversões

- **int(a)**: converte um número real ou string em um inteiro.

```
>>> int(12.67)
12
>>> int('12')
12
```

- **float(a)**: converte um inteiro ou string em um real.

```
>>> float('5')
5.0
>>> float('10')
10.0
```

- **str(a)**: converte um inteiro, complexo ou real em uma string.

```
>>> str(12.5)
'12.5'
>>> str(12)
'12'
>>> str(12+4j)
'(12+4j)'
```

Funções Pré-definidas: conversões

- **complex(a)**: converte uma string, inteiro ou real em um complexo.

```
>>> complex(2)
(2+0j)
>>> complex('12')
(12+0j)
>>> complex('12.1')
(12.1+0j)
>>> complex(6.5)
(6.5+0j)
```

- **list(a)** e **tuple(a)**: converte uma string em uma lista e uma tupla, respectivamente.

```
>>> list('abacate')
['a', 'b', 'a', 'c', 'a', 't', 'e']
>>> tuple('abacate')
('a', 'b', 'a', 'c', 'a', 't', 'e')
```

Funções Pré-definidas: leitura do teclado

- **raw_input([prompt]):** lê uma string do teclado. Se prompt for declarado, ele será impresso sem alimentação de linha.

```
>>> a = raw_input("Entre com o seu nome: ")
Entre com o seu nome: Alberto
>>> a
'Alberto'
```

- **input([prompt]):** lê qualquer coisa do teclado. Strings devem ser protegidas com aspas simples ou duplas. O comando input é o mesmo que eval(raw_input(prompt)).

```
>>> a = input("Entre com algo: "); a
Entre com algo: { 'pai' : 'João', 'idade' = 60 }
{'idade': 70, 'pai': 'Jo\xe3o'}
>>> a = input("Entre com algo: "); a
Entre algo com algo: 'Alberto Santos Dumont'
'Alberto Santos Dumont'
>>> a = input("Entre com algo: "); a
Entre algo com algo: 5 + 4
9
```


Docstring: Documentação

O Python possui suporte nativo à documentação de código. Strings de documentação, chamadas de docstrings, são adicionados ao início de módulos, funções e classes, para instruir o funcionamento e funcionalidades dos módulos, funções e classes.

Estas docstrings podem ser escritas entre aspas duplas:

```
"mensagens de uma única linha"
```

ou três aspas duplas:

```
""" mensagem com mais  
que uma linha devem ser  
escritas entre três aspas  
duplas """
```

Docstring: Documentação

```
#-*- coding: iso-8859-1 -*-
# Módulo Sequência de Fibonacci: fibonacci.py
""" Modulo Fibonacci
    Funcoes:
        fib(n)
        fib2(n)
"""
def fib(n):
    " Write Fibonacci series up to n "
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n):
    " Return Fibonacci series up to n "
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Docstring: Documentação

Docstrings são extremamente úteis para documentar módulos, funções e classes em seu código, o que facilita consultas futuras destes módulos e funções. Veja o exemplo a seguir:

```
>>> import fibonacci
>>> print fibonacci.__doc__
Modulo Fibonacci"
    Funcoes:
        fib(n)
        fib2(n)

>>> print fibonacci.fib.__doc__
Write Fibonacci series up to n
>>> print fibonacci.fib2.__doc__
Return Fibonacci series up to n
```

```
print open.__doc__
open(name[, mode[, buffering]]) -> file object
```

Open a file using the file() type, returns a file object.

Manipulação de Arquivos: open()

A função `open` retorna um objeto da classe `file`, que permite fazer a leitura e escrita em arquivos das mais diversas formas. Na sua forma padrão, o arquivo é aberto somente para leitura:

```
f = open("/etc/fstab")
>>> print f
<open file '/etc/fstab', mode 'r' at 0x7f800b16de00>
```

Para conhecer todos os métodos do objeto arquivo você pode utilizar o comando `dir(f)`:

```
>>> dir(f)
['__class__', '__delattr__', '__doc__', '__enter__',
'__exit__', '__getattr__', '__hash__', '__init__',
'__iter__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__str__', 'close', 'closed',
'encoding', 'fileno', 'flush', 'isatty', 'mode', 'name',
'newlines', 'next', 'read', 'readinto', 'readline',
'readlines', 'seek', 'softspace', 'tell', 'truncate', 'write',
'writelines', 'xreadlines']
```

Manipulação de Arquivos: open() - uma dica

A documentação de cada método pode ser muito útil para se aprender como utilizá-los:

```
>>> print f.read.__doc__  
read([size]) -> read at most size bytes, returned as a string.
```

If the size argument is negative or omitted, read until EOF is reached. Notice that when in non-blocking mode, less data than what was requested may be returned, even if no size parameter was given.

Lendo 10 *bytes*

```
>>> print '>>%s<<' % f.read(10)  
>>/dev/sda5 <<
```

Manipulação de Arquivos: open()

Sintaxe completa do file:

`open(name[, mode[, buffering]]) -> file object`

Abre um arquivo no modo 'r' (leitura - modo padrão), 'w' (escrita), ou 'a' (append). O arquivo será criado se não existir, quando aberto no modo 'w' ou 'a'.

Adicione um 'b' para abrir um arquivo no modo binário, e um '+' para permitir escrita e leitura simultaneamente.

Se buffering for:

- 0, nenhum buffer será alocado para o arquivo;
- 1, um buffer de uma linha será alocado;
- nn, um buffer de nn bytes será alocado para o arquivo.

Se um 'U' (*universal new linw suport*) for adicionado ao modo, todo fim de linha será visto como um '\n'. 'U' não pode ser usado com os modos 'w' ou '+'.

Manipulação de Arquivos: `open()` ou `file()`

Alguns métodos de `open()`:

- **`read([size])`**: ler `size` bytes e retorna como uma string. Se `size` for negativo ou omitido a leitura será feita até alcançar o final do arquivo.

```
>>> f.read()
'Em 23 de outubro de 1906, voou cerca de 60 metros e a uma
altura de dois a tr\xeaxas metros com seu 14 Bis, no Campo de
Bagatelle em Paris.\nMenos de um m\xeaxas depois, em 12 de
novembro, repetiu o feito e, diante de uma multid\xee3o de
testemunhas, percorreu 220 metros a uma altura de 6 metros.
\n0 v\xef4o do 14-Bis foi o primeiro verificado pelo
Aeroclube ... \n'
```

Manipulação de Arquivos: open()

- **seek(offset[, whence])**: move o ponteiro de acesso ao arquivo para uma nova posição. O offset é contado em bytes, 0 para o início do arquivo. O argumento opcional whence pode assumir três valores:
 - 0 - padrão, mover para o início do arquivo;
 - 1 - fica no mesmo lugar;
 - 2 - move para o final do arquivo.

```
>>> f.read()                # já alcançou o final do arquivo
''

>>> f.seek(0)               # alcança o início do arquivo
>>> f.read(200)              # ler 200 bytes
'Em 23 de outubro de 1906, voou cerca de 60 metros e a uma
altura de dois a tr\xeas metros com seu 14 Bis, no Campo de
Bagatelle em Paris.\nMenos de um m\xeas depois, em 12 de
novembro, repetiu o feito e, dia'
>>> f.seek(0, 2)             # vai para o final do arquivo
>>> f.read(200)
''
```


Manipulação de Arquivos: open()

- **readline([size]):** ler a próxima linha, como uma string. Se size for diferente de zero, isto irá restringir o comprimento de bytes lidos.

```
>>> f.seek(0)
>>> f.readline()
'Em 23 de outubro de 1906, voou cerca de 60 metros e a uma
altura de dois a tr\xeaxas metros com seu 14 Bis, no Campo de
Bagatelle em Paris.\n'
>>> f.readline(100)
'Menos de um m\xeaxas depois, em 12 de novembro, repetiu o feito
e, diante de uma multitud\xee3o de testemunhas,'
>>> f.readline(100)
' percorreu 220 metros a uma altura de 6 metros. \n'
>>> f.readline()
'0 v\xef4o do 14-Bis foi o primeiro verificado pelo
Aeroclube ... \n'
```

Se não quiser ver os caracteres de controle, use: **print f.readline()** ao invés de apenas **f.readline()**.

Manipulação de Arquivos: open()

- **close()**: fecha um arquivo.

```
>>> f.close()
>>> f.readline()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file
```

- **write(str)**: escreve a string str no arquivo. Devido ao buffering, pode ser necessário usar os métodos flush() ou close(), para que o arquivo no disco reflita as alterações feitas.

```
>>> f = file('texto.txt', 'r+')
>>> f.seek(0, 2)                # avança para o final do arquivo
>>> f.write('Retirado http://pt.wikipedia.org/wiki/Santos\_Dumont\n')
>>> f.seek(0)
>>> print f.read()
Em 23 de outubro de 1906, voou cerca de 60 metros e a uma altura de
dois a três metros com seu 14 Bis, no Campo de Bagatelle em Paris.
Menos de um mês depois, em 12 de novembro, repetiu o feito e, diante
de uma multidão de testemunhas, percorreu 220 metros a uma altura de 6
metros.
O vôo do 14-Bis foi o primeiro verificado pelo Aeroclube ...
Retirado http://pt.wikipedia.org/wiki/Santos\_Dumont
```

Manipulação de Arquivos: open()

- **tell()**: apresenta a posição corrente, para escrita e leitura, no arquivo.

```
>>> f.seek(0, 2); end = f.tell()
>>> f.seek(0); pos = f.tell()
>>> line = 1
>>> while pos != end:
...     print 'Linha %d - posição %d' % (line, pos)
...     line += 1
...     l = f.readline()
...     pos = f.tell()
...
Linha 1 - posição 0
Linha 2 - posição 135
Linha 3 - posição 284
Linha 4 - posição 345
```

Manipulação de Arquivos: open()

- **flush():** grava as alterações em buffer, no arquivo.

```
>>> f.flush()
```

- **mode:** retorna a string de modo, com que o arquivo foi aberto.

```
>>> f.mode  
'r+'
```

- **name:** retorna uma string com o nome do arquivo foi aberto.

```
>>> f.name  
'texto.txt'
```

- **closed:** retorna True se o arquivo estiver fechado e False se estiver aberto.

```
>>> f.closed  
False  
>>> f.close()  
>>> f.closed  
True
```


Manipulação de Arquivos: open()

- **next()**: ler o próximo valor do arquivo corrente ou StopIteration, se o final do arquivo for alcançado. A menos do StopIteration, gerado ao alcançar o final do arquivo, o método next() retorna o mesmo que o readline().

```
>>> f.seek(0); f.next()
'Em 23 de outubro de 1906, voou cerca de 60 metros e a uma
altura de dois a tr\xeeas metros com seu 14 Bis, no Campo de
Bagatelle em Paris.\n'
>>> f.next()
...

>>> f.next()
'Retirado http://pt.wikipedia.org/wiki/Santos_Dumont\n'
>>> f.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Orientação a Objetos

“Orientação a objetos, OO, é um termo que descreve uma série de técnicas para estruturar soluções para problemas computacionais. é um paradigma de programação no qual um programa é estruturado em objetos, e que enfatiza os aspectos abstração, encapsulamento, polimorfismo e herança.”

Objetos: Como observamos em seções anteriores, em Python tudo é um objeto, com atributos e métodos: valores, tipos, classes, funções, métodos, ...

Orientação a Objetos: classes

Classes: A estrutura fundamental para definir uma objeto é a classe.

Usarei o exemplo do Christian R. Reis. Vamos criar um módulo chamado `formas.py` com a definição de uma classe *Retângulo*.

Esta classe possuirá dois atributos: `lado_a` e `lado_b`, e classe irá calcular área e perímetro com os métodos *calcula_area* e *calcula_perimetro*:

```
# Define formas
#-*- coding: iso-8859-1 -*-

class Retangulo:
    lado_a = None
    lado_b = None

    def __init__(self, lado_a, lado_b):
        self.lado_a = lado_a
        self.lado_b = lado_b
        print 'Criando uma nova instância retângulo.'

    def calcula_area(self):
        return self.lado_a * self.lado_b

    def calcula_perimetro(self):
        return 2 * (self.lado_a + self.lado_b)
```


Orientação a Objetos: classes

Esta classe possui três métodos, sendo um deles um método especial `__init__()`. Este é o método construtor padrão do Python, invocado quando uma classe é **instanciada** (nome dado a criação de um objeto a partir de uma classe). Este método é opcional.

Observe que todos os métodos possuem como atributo a variável *self*, que é manipulada no interior do método. Em Python, o primeiro argumento é especial, sendo seu nome por convenção *self*.

Instâncias: A instância é o objeto criado com base em uma classe definida. Uma descrição abstrata da dualidade classe-instância:

- a classe é apenas uma matriz, que especifica os objetos, mas que não pode ser utilizada diretamente;
- a instância representa o objeto concretizado a partir de uma classe.

Orientação a Objetos: classes

Agora vamos brincar um pouco com nossa primeira classe, do módulo formas.py:

```
>>> from formas import * # importa apenas Retangulo, por hora
>>> r1 = Retangulo(2, 5)
Criando uma nova instância Retângulo
>>> r2 = Retangulo(3, 4)
Criando uma nova instância Retângulo
```

```
def __init__(self, lado_a, lado_b):
    self.lado_a = lado_a
    self.lado_b = lado_b
    print 'Criando uma nova instância retângulo.'
```

Orientação a Objetos: classes

Após instanciados os objetos `r1` e `r2`, os métodos *calcula_area* e *calcula_perimetro* são disponibilizados

```
>>> r1.calcula_perimetro()
14
>>> r2.calcula_area()
12
```

```
def calcula_area(self):
    return self.lado_a * self.lado_b

def calcula_perimetro(self):
    return 2 * (self.lado_a + self.lado_b)
```

```
>>> r1.lado_a
2
>>> r1.lado_b
5
```


Orientação a Objetos: classes

Atributos privados e protegidos: Python não possui uma construção sintática para definir atributos como privados em uma classe, mas existem formas de se indicar que um atributo não deve ser acessado externamente:

- Por convenção, atributos iniciados por um sublinhado, “_”, não devem ser acessados externamente;

```
>>> class test:
...     atr1 = "atributo publico"
...     _atr2 = "atributo privado"
...
>>> a = test
>>> a.atr1
'atributo publico'
>>> a._atr2
'atributo privado'
```

Orientação a Objetos: classes

- Suporte no próprio interpretador: atributos iniciados por dois sublinhados, “__”, são renomeados para prevenir de serem acessados externamente.

```
>>> class test:
...     atr1 = "atributo publico"
...     __atr2 = "atributo privado"
...
>>> a = test
>>> a.atr1
'atributo publico'
>>> a.__atr2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: class test has no attribute '__atr2'
```

Na verdade o atributo não está inacessível, apenas teve seu nome alterado para `_test__atr2`

```
>>> a._test__atr2
'atributo privado'
```

Orientação a Objetos: herança

Herança é uma forma de derivar classes novas a partir de classes-bases.

Python suporta herança simples e herança múltiplas. A sintaxe de uma classe é:

```
class nome-classe(base_1, base_2, ..., base_n):  
    atributo_1 = valor_1  
    ...  
    atributo_z = valor_z  
  
    def nome_método_1(self, arg_1, arg_2, ..., arg_k):  
        # bloco de comandos do método_1  
        ...  
  
    def nome_método_m(self, arg_1, arg_2, ..., arg_n):  
        # bloco de comandos do método_m  
        ...
```


Orientação a Objetos: herança

Vamos definir a classe *Quadrado*, como derivada de *Retangulo*.
Adicione o texto abaixo ao *formas.py*:

```
class Quadrado(Retangulo):  
    def __init__(self, lado):  
        self.lado_a = self.lado_b = lado
```

Em seguida vamos criar um quadrado de aresta 10 e ver seus métodos e atributos

```
>>> from formas import *  
>>> r1= Quadrado(10)  
>>> dir(r1)  
['__doc__', '__init__', '__module__', 'calcula_area',  
'calcula_perimetro', 'lado_a', 'lado_b']  
>>> r1.lado_a, r1.lado_b  
(10, 10)  
>>> r1.calcula_area(), r1.calcula_perimetro()  
(100, 40)
```

Orientação a Objetos: herança

Invocando métodos de uma classe-base: poderíamos ter criado a classe *Quadrado* utilizando o construtor da classe *Retangulo*, invocando-o de dentro da classe *Quadrado*. Vamos fazer isto para uma classe *Square*:

```
>>> from formas import *
>>> class Square(Retangulo):
...     def __init__(self, lado):
...         Retangulo.__init__(self, lado, lado)
...
>>> r1 = Square(10)
Criando uma nova instância retângulo.
>>> r1.lado_a, r1.lado_b
(10, 10)
>>> r1.calcula_area(), r1.calcula_perimetro()
(100, 40)
```

Orientação a Objetos: herança

Uma fraqueza? Alguns cuidados devem ser tomados ao se alterar atributos em um objeto. Observe o exemplo abaixo:

```
>>> class Foo:
...     a = [5, 3]
...
>>> h = Foo()
>>> h.a.append(2); h.a
[5, 3, 2]
>>> g = Foo()
>>> g.a
[5, 3, 2]
```

“Esta particularidade é frequentemente fonte de bugs difíceis de localizar, e por este motivo se recomenda fortemente que não se utilize variáveis de tipos mutáveis em classes.” [Christian Reis]

Isto ocorre sempre com atributos mutáveis como listas e dicionários. Para atributos não mutáveis, as atribuições são sempre feitas na variável da instância local, e não da classe, como era de se esperar.

```
>>> class Foo2:
...     a = 1
...
>>> h = Foo2()
>>> h.a = 2; h.a
2
>>> g = Foo2()
>>> g.a
1
```


Orientação a Objetos: herança

Façamos alguns testes agora:

```
>>> j = Foo2()  
>>> j.a  
5  
>>> j.set_a(8)  
>>> k = Foo2()  
>>> j.a, k.a  
(8, 5)
```

Desta forma não há superposição na definição da classe. Este é o custo da alta flexibilidade do Python.

Enquanto por um lado ele lhe permite alterar a definição de um atributo de uma classe em tempo de execução, por outro lado, pode gerar erros de difícil localização.

Orientação a Objetos: funções úteis

Vamos ver duas funções importantes para conhecer a hierarquia de uma classe e instâncias

- **isinstance(objeto, classe):** verifica se o objeto passado é uma instância da classes

```
>>> from formas import *
>>> f1 = Quadrado(15)
>>> isinstance(f1, Quadrado)
True
>>> isinstance(f1, Retangulo)
True
```

- **issubclass(classe_a, classe_b):** verifica se classe_a é uma sub-classe de classe_b

```
>>> issubclass(Quadrado, Retangulo)
True
>>> issubclass(Retangulo, Quadrado)
False
```

Orientação a Objetos: funções úteis

- **hasattr(objeto, atributo):** verifica se um objeto possui um atributo.

```
>>> hasattr(f1, lado_a)
True
>>> hasattr(f1, lado)
False
```

Uma função para verificar se um objeto é uma forma:

```
>>> def IsForma(obj):
...     return hasattr(obj, 'lado_a') and hasattr(obj,
...         'lado_b')
...
>>> IsForma(f1)
True
>>> a = 5
>>> IsForma(a)
False
```


Orientação a Objetos

Introspecção e reflexão: Python permite obter, em tempo de execução, informações a respeito do tipo dos objetos, incluindo informações sobre a hierarquia de classes.

- **dir(objeto):** permite conhecer todos os atributos e métodos de uma classe ou instância.

```
>>> from formas import *
>>> r = Quadrado(13)
>>> dir(r)
['__doc__', '__init__', '__module__', 'calcula_area',
'calcula_perimetro', 'lado_a', 'lado_b']
```

- **__class__:** este atributo da instância armazena o seu objeto classe correspondente.

```
>>> r.__class__
<class formas.Quadrado at 0x2b717bc48770>
```

Orientação a Objetos

- **`__dict__`**: apresenta um dicionário com todos os atributos de uma instância.

```
>>> r.__dict__  
{'lado_a': 13, 'lado_b': 13}
```

- **`__class__`**: este atributo da instância armazena o seu objeto classe correspondente.

```
>>> r.__class__  
<class formas.Quadrado at 0x2b717bc48770>
```

- **`__module__`**: apresenta uma string o nome do módulo o qual a instância ou a classe foi importada.

```
>>> r.__module__  
'formas'
```

Orientação a Objetos

- **Classe__bases__**: apresenta uma tupla com as classes herdadas por Classe.

```
>>> Quadrado.__bases__  
(<class formas.Retangulo at 0x2abc1ac8a710>,)
```

- **Classe__name__**: apresenta uma string com o nome da classe.

```
>>> Quadrado.__name__  
'Quadrado'
```


Importando Módulos

Nesta última seção do curso, vou dedicar a apresentação de alguns módulos padrões do Python.

Estes módulos são carregados com o comando **import**, como já foi apresentado ao longo desta apresentação.

As suas sintaxes básicas são:

```
import <módulo_1> [ as nome_1 ] [, <módulo_2> [ as  
nome_2]] ...
```

```
from <módulo> import [<ident_1>, <ident_2>, ...]
```

Na segunda forma, ainda é possível usar ***** para indicar a importação de todos os métodos, funções, ..., para a raiz.

```
from <módulo> import *
```

Importando Módulos

Exemplos:

```
>>> import fibonacci
>>> fibonacci.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> dir()
['__builtins__', '__doc__', '__name__', 'fibonacci']
```

Importando tudo para a raiz:

```
>>> from fibonacci import *
>>> fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> dir()
['__builtins__', '__doc__', '__name__', 'fib', 'fib2']
```

Nas seções seguintes, irei apresentar diversos módulos padrões do Python. Para conhecer todos os métodos, funções, ... de um módulo use o comando: `dir(módulo)`

Módulo - sys

sys: Este módulo possui várias funções que permitem interagir com o próprio interpretador Python:

- **ps1 e ps2:** definem os prompts do Python ("**>>>**" e "**...**").

```
>>> import sys
>>> sys.ps1 = '> '
> sys.ps2 = '. '
> for i in range(10)
.     print i
```

- **argv:** armazena os argumentos passados pela linha de comandos na lista de strings `argv[]`, onde o primeiro elemento é o nome do programa chamado, seguido pelos outros argumentos. Por exemplo, considere um módulo Python `args.py`, com o conteúdo:

```
# Modulo args.py
from sys import argv
print sys.argv
```


Módulo - sys

```
$ python args.py 2 5 -3  
['args.py', '2', '5', '-3']
```

- **path:** apresenta os caminhos utilizados pelo Python para buscar os módulos solicitados pelo comando **import**.

```
>>> sys.path  
['', '/usr/lib64/python25.zip', '/usr/lib64/python2.5',  
'/usr/lib64/python2.5/plat-linux2', '/usr/lib64/python2.5/lib-  
tk', '/usr/lib64/python2.5/lib-dynload',  
'/usr/lib64/python2.5/site-packages',  
'/usr/lib64/python2.5/site-packages/gtk-2.0']
```

- **platform, prefix, version, ...:** informações sobre o Python parâmetros de sua instalação.

```
>>> sys.platform, sys.prefix, sys.version  
('linux2', '/usr', '2.5.1 (r251:54863, Sep  4 2007,  
19:00:19) \n[GCC 4.1.2]')
```

Módulo - sys

- **stdin, stdout, stderr:** entrada, saída e saída de erro padrões. Permite redirecionar as entradas e saídas padrões do sistema.

```
>>> sys.stdout.write('Hello World')  
Hello World>>>
```

- **exit:** encerra uma seção do Python mais diretamente

```
>>> sys.exit()  
$ _
```

Módulo - re

re: (*regular expression*) este módulo fornece ferramentas para filtrar strings através de Expressões Regulares.

- **findall:** permite encontrar a ocorrência de uma string, filtrando-a por uma expressão regular.

```
>>> import re
>>> re.findall(r'\b[a-z]*', 'which foot or hand fell
fastest')
['foot', 'fell', 'fastest']
```

- **sub:** substitui uma ocorrência de uma string por outra.

```
>>> re.sub(r'\bAMD', r'AuthenticAMD', 'AMD Turion(tm) 64 X2
Mobile')
'AuthenticAMD Turion(tm) 64 X2 Mobile'
```


Módulo - re

Substituindo duas ocorrências de uma string por uma.

```
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')  
'cat in the hat'
```

Sem a mesma eficiência, o mesmo poderia ser feito com o método `replace` de string.

```
>>> 'cat the the hat'.replace('the the', 'the')  
'cat the hat'
```

```
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat cat the the hat hat,  
and my my shoes')  
'cat the hat, and my shoes'
```

Módulo - math

math: este módulo fornece acesso a diversas as funções matemáticas e constantes.

- **sqrt, cos, sin, ...** : diversas funções matemáticas. As funções ausentes podem ser construídas a partir destas.

```
>>> dir(math)
['__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']
```

```
>>> import math
>>> def Sin(a):
...     "Calcula seno de angulo em graus"
...     ang = a*math.pi/180.    # mesmo que radians()
...     return math.sin(ang)
...
>>> Sin(30)
0.49999999999999994
>>> Sin(60)
0.8660254037844386
```

Módulo - random

random: este módulo permite gerar números aleatórios, sorteios, seqüências, distribuições reais, uniformes, gamma, ... Veja a documentação para mais detalhes:

```
>>> print random.__doc__
```

- **choice(lista):** escolhe de forma aleatória um elemento de uma lista

```
>>> import random
>>> random.choice(['goiaba', 'laranja', 'abacate', 'pera'])
'pera'
```

- **randrange(n):** gera um inteiro aleatório entre 0 e n-1

```
>>> random.randrange(10)
3
```

- **randint(n, m):** gera um inteiro aleatório entre n e m, incluindo os extremos, m e n.

```
>>> random.randint(3, 6)
6
```


Módulo - random

- **sample(lista, n):** gera uma lista com n elementos da lista, sem repetição dos elementos. O número de elementos sorteados, m, deve ser menor ou igual ao comprimento da lista.

```
>>> from random import sample
>>> sample([0, 1, 2, 3], 4)
[1, 3, 0, 2]
>>> def sena(n = 1):
...     """Imprime n sorteios para a MegaSena"""
...     print "Sorteios da MegaSena"
...     for i in xrange(n):
...         print str(i+1) + ": " + str(sample(xrange(60), 6))
...
>>> sena(5)
Sorteios da MegaSena
1: [30, 31, 52, 3, 58, 49]
2: [20, 46, 1, 6, 30, 12]
3: [14, 39, 54, 57, 42, 15]
4: [48, 36, 33, 5, 3, 23]
5: [13, 53, 6, 25, 37, 55]
```

Módulos para Internet - urllib2 e smtplib

urllib2: este módulo permite criar navegar pela internet, carregar páginas, pesquisar, ...

```
>>> import urllib2
>>> for line in
urllib2.urlopen('http://tycho.usno.navy.mil/cgi-
bin/timer.pl'):
...     if 'EST' in line:         # look for Eastern Standard
Time
...         print line
<BR>Nov. 25, 09:43:32 PM EST
```

Módulos para Internet - urllib2 e smtplib

smtplib: com este módulo é possível enviar emails através de um servidor smtp.

```
>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org',
'jcaesar@example.org',
"""To: jcaesar@example.org
From: soothsayer@example.org
Beware the Ides of March.
""")
>>> server.quit()
```


Módulo - datetime

datetime: este módulo fornece classes para manipulação de datas e horas nas mais variadas formas.

- **date(ano, mês, dia):** cria um objeto data.

```
>>> from datetime import date
>>> hoje = date.today()
>>> nascimento = date(1986, 5, 16)
>>> idade = hoje - nascimento
>>> print "Sua idade é %d anos" % int(idade.days/365)
Sua idade é 11 anos
>>>
>>> dir(date)
['__add__', '__class__', '__delattr__', '__doc__', '__eq__',
'__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__le__', '__lt__', '__ne__', '__new__',
'__radd__', '__reduce__', '__reduce_ex__', '__repr__',
'__rsub__', '__setattr__', '__str__', '__sub__', 'ctime',
'day', 'fromordinal', 'fromtimestamp', 'isocalendar',
'isoformat', 'isoweekday', 'max', 'min', 'month', 'replace',
'resolution', 'strftime', 'timetuple', 'today', 'toordinal',
'weekday', 'year']
```

Módulo - zlib

zlib: este módulo permite trabalhar com dados comprimidos, comprimindo, descomprimindo, ...

- **compress(string), decompress(string):** comprime e descomprime uma string

```
>>> from zlib import compress, decompress, crc35
>>> s = "Em 23 de outubro de 1906, voou cerca de 60 metros e a
uma altura de dois a três metros com seu 14 Bis, no Campo de
Bagatelle em Paris."
>>> len(s)
134
>>> z = compress(s)
>>> len(z)
113
```

- **crc32(string):** computa o CRC-32 de uma string (checksum)

```
>>> from zlib import crc35
>>> crc32(s)
3810011638
```


Módulo - timeit

timeit: este módulo permite monitorar o desempenho na execução de comandos Python

```
>>> Timer('xrange(100)').timeit()  
0.93808984756469727  
>>>  
>>> Timer('range(100)').timeit()  
2.9305579662322998
```

Observe que o `xrange(100)` chega a ser 68% mais rápido que o `range(100)`. Isto mostra a vantagem em se usar o `xrange` em loops e outras ocasiões.

```
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()  
0.38922691345214844  
>>> Timer('a, b = b, a', 'a=1; b=2').timeit()  
0.31212997436523438
```

No swap acima foi possível conseguir 20% a menos no processamento, evitando-se uma atribuição em uma variável temporária.

Considerações Finais

Existe ainda muitos aspectos, módulos, funções, objetos, funções, ... disponíveis para o Python.

Bibliografia:

- Tutorial Python, release 2.4.2 de Guido van Rossum, Fred L . Drake, Jr., editor, tradução: Python Brasil
- Python na Prática - Um curso objetivo de programação em Python, <http://www.async.com.br/projects/pnp/>, de Christian Robottom Reis, Async Open Source, kiko@async.com.br
- Documentação do Python (dir() e __doc__)