Python

Uma visão geral sobre a linguagem

Guilherme Artém
Henrique Castro
José Mauro

Sumário

- Introdução
- Um primeiro programa
- Compilação e Interpretação
- Sintaxe
  - Blocos
  - Objetos
- Controle de Fluxo
- Estruturas de Repetição
  - For
  - While
- Tipos
  - Numéricos
  - Textos
  - Listas

Sumário

Sumário

- Herança Simples
- Herança Múltipla
- Sobrecarga de Operadores
  - Coleções
- Concorrência
- Avaliação de LP
- Mais informações

Introdução

- Python é uma linguagem de programação poderosa e fácil de aprender.
- É uma Linguagem de programação com suporte a vários paradigmas:
  - Estruturado.
  - Orientado a Objetos.
  - Funcional.
- É completamente Open Source.
- Possui uma sintaxe legível.
- É uma linguagem interpretada e interativa.
- Possui tipagem dinâmica (não é necessário declarar tipos).

Introdução

- Faz o programador se concentrar na resolução do problema (questões como compilação, declaração de variáveis, sintaxe complexa, etc).
- Possibilidade de integração com código C, C++, Java (Jython), etc.
- Ótima linguagem para prototipagem, RAD (Rapid Application Development) extensão e integração de aplicações e componentes, scripts.
- Compilação em tempo de execução (on the fly).
- É uma linguagem de tipagem forte.

Introdução

- A linguagem é fortemente tipada no sentido que erros de tipagem são impedidos de acontecer em tempo de execução.
- A quantidade de conversões de tipo é baixa, porém não há uma checagem “estática” de tipos;
- A inexistência desta checagem aumenta a velocidade de compilação, importante para uma linguagem interpretada.
- “Duck Typing”

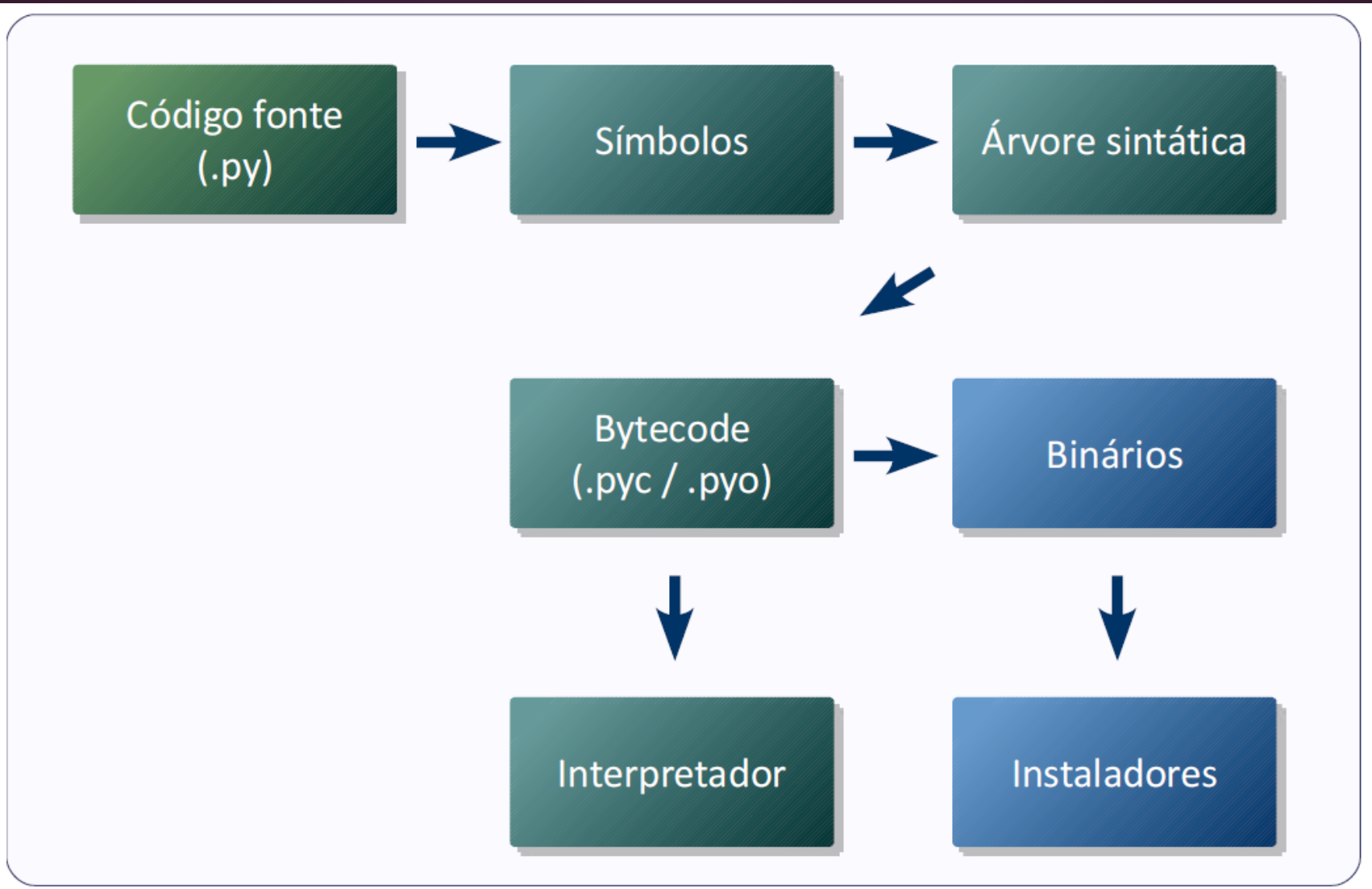
Um primeiro Programa

- Hello World (Python 2.7.1):

```
>>> print("Hello World!")
Hello World!
>>> x=5
>>> print(x)
5
>>> x="Hello World"
>>> print(x)
Hello World
```

Compilação e Interpretação

- O código fonte é traduzido pelo Python para *bytecode*, que é um formato binário com instruções para o interpretador. O *bytecode* é multiplataforma e pode ser distribuído e executado sem fonte original.



Como é feita a compilação e interpretação

Compilação e Interpretação

- Por padrão, o interpretador compila o código e armazena o *bytecode* em disco, para que a próxima vez que o executar, não precise compilar novamente o programa, reduzindo o tempo de carga na execução.
- Se os arquivos fontes forem alterados, o interpretador se encarregará de regerar o *bytecode* automaticamente.

Compilação e Interpretação

- Devido a natureza de Python como linguagem interpretada, todas as amarrações que não são feitas em tempo de projeto de LP (palavras reservadas, tamanho de um inteiro) são executadas em tempo de execução.

Sintaxe

- Um programa feito em Python é constituído de linhas, que podem continuar nas linhas seguintes, pelo uso do caractere de barra invertida (\) ao final da linha ou parênteses, colchetes ou chaves, em expressões que utilizam tais caracteres.
- O caractere # marca o inicio de comentário. Qualquer texto depois do # será ignorado até o fim da linha , com exceção dos comentários funcionais.

Sintaxe

- Comentários funcionais são usados para:
  - Alterar a codificação do arquivo fonte do programa acrescentando um comentário com o texto “#-*- coding: <encoding> -*#-” no inicio do arquivo.
  - Definir o interpretador que será utilizado para rodar o programa em sistemas UNIX, através de um comentário começando com “#!” no inicio do arquivo, que indica o caminho para o interpretador.

Sintaxe

- Comentários funcionais são usados para:
  - Alterar a codificação do arquivo fonte do programa acrescentando um comentário com o texto “#-*- coding: <encoding> -*#-” no inicio do arquivo.
  - Definir o interpretador que será utilizado para rodar o programa em sistemas UNIX, através de um comentário começando com “#!” no inicio do arquivo, que indica o caminho para o interpretador.

```
# -*- coding: latin1 -*-
# Uma linha quebrada por contra-barra
a = 7 * 3 + \
5 / 2

# Uma lista (quebrada por vírgula)
b = ['a', 'b', 'c',
'd', 'e']
```

Exemplo de linhas quebradas

```
#!/usr/bin/env python
# -*- coding: latin1 -*-

# Uma linha de código que mostra o resultado de 7 vezes 3
print 7 * 3
```

Exemplo de comentários funcionais

Sintaxe

- Palavras reservadas:

Keywords

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

- Tokens inválidos (apenas válidos dentro de strings):
  - $?
  - @ antes do python 2.4

Sintaxe

- Funções pré-definidas:

		Built-in Functions		
abs()	divmod()	input()	open()	staticmethod()
all()	enumerate()	int()	ord()	str()
any()	eval()	isinstance()	pow()	sum()
basestring()	execfile()	issubclass()	print()	super()
bin()	file()	iter()	property()	tuple()
bool()	filter()	len()	range()	type()
bytearray()	float()	list()	raw_input()	unichr()
callable()	format()	locals()	reduce()	unicode()
chr()	frozenset()	long()	reload()	vars()
classmethod()	getattr()	map()	repr()	xrange()
cmp()	globals()	max()	reversed()	zip()
compile()	hasattr()	memoryview()	round()	__import__()
complex()	hash()	min()	set()	apply()
delattr()	help()	next()	setattr()	buffer()
dict()	hex()	object()	slice()	coerce()
dir()	id()	oct()	sorted()	intern()

Sintaxe

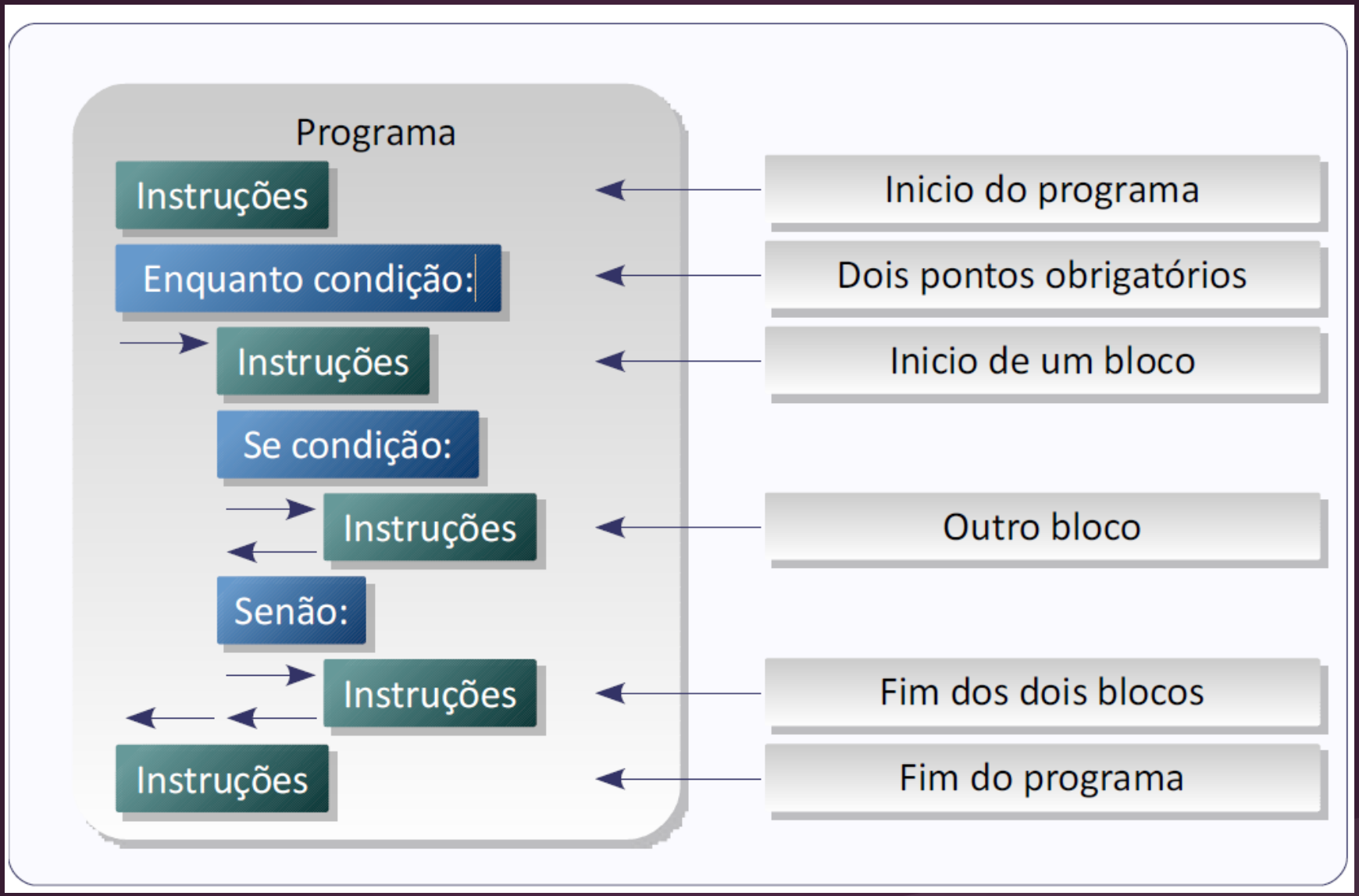
- Operadores:

Operators and their evaluation order		
Highest	Operator	Comment
	, [...] {...} `...`	Tuple, list & dict. creation; string conv.
	s[i] s[i:j] s.*attr* f(...)	indexing & slicing; attributes, function calls
	+x, -x, ~x	Unary operators
	x**y	Power
	x*y x/y x%y	mult, division, modulo
	x+y x-y	addition, substraction
	x<<y x>>y	Bit shifting
	x&y	Bitwise "and"; also intersection of sets
	x^y	Bitwise exclusive or
	x|y	Bitwise "or"; also union of sets
	x<y x<=y x>y x>=y x==y x!=y x<>y x **is** y x **is not** y x **in** s x **not in** s	Comparison, identity, membership
	not x	boolean negation
	x **and** y	boolean and
	x **or** y	boolean or
Lowest	**lambda** *args*: *expr*	anonymous function

  - Maior parte dos operadores pode ser sobrescrita;
  - Será explicado em slides futuros;
  - Normalmente um operador tem uma segunda representação para ser sobrescrito;
    - ex: ‘+’, para sobrescrever usa-se ‘__add__’.

Blocos

- Em Python, os blocos de código são delimitados pelo uso de indentação, que deve ser constante no bloco de código.
- É considerada uma boa prática para manter a consistência no projeto todo e evitar a mistura de tabulações e espaços.
- Aumenta a legibilidade, diminui a redigibilidade.



Como endentar um programa em Python

Objetos

- Python é uma linguagem orientada a objeto, sendo assim as estruturas de dados possuem atributos (os dados em si) e métodos (rotinas associadas aos dados).
- Tanto os atributos quanto os métodos são acessados usando ponto (.).

Objetos

- Para mostrar um atributo:
  - print objeto.atributo
- Para executar um método:
  - objeto.metodo(argumentos)
- Mesmo um método sem argumentos precisa de parenteses:
  - objeto.metodo()

```
if <condição>:
    <bloco de código>
elif <condição>:
    <bloco de código>
elif <condição>:
    <bloco de código>
else:
    <bloco de código>
```

Controle de Fluxo

- <condição>: sentença que possa ser avaliada como verdadeira ou falsa.
- <bloco de código>: sequência de linhas de comando.
- As clausulas elif e else são opcionais e podem existir vários elifs para o mesmo if, porém apenas um else ao final.
- Parênteses só são necessários para evitar ambiguidades.

```
temp = int(raw_input('Entre com a temperatura: '))

if temp < 0:
    print 'Congelando...'
elif 0 <= temp <= 20:
    print 'Frio'
elif 21 <= temp <= 25:
    print 'Normal'
elif 26 <= temp <= 35:
    print 'Quente'
else:
    print 'Muito quente!'
```

Exemplo Controle de Fluxo

Estruturas de Repetição

- **For:** É a estrutura de repetição mais usada no Python. A instrução aceita não só sequências estáticas, mas também sequências geradas por iteradores.
  - Iteradores são estruturas que permitem iterações, ou seja, acesso aos itens de uma coleção de elementos, de forma sequencial.

```
for <referência> in <sequência>:
    <bloco de código>
    continue
    break
else:
    <bloco de código>
```

Laço For

Estruturas de Repetição

- A clausula *break* interrompe o laço e *continue* passa para a próxima iteração. O código dentro do *else* é executado ao final do laço, a não ser que o laço tenha sido interrompido por *break*.
  - A função *range(m, n, p)*, é muito útil em laços, pois retorna uma lista de inteiros, começando em *m* e menores que *n*, em passos de comprimento *p*, que podem ser usados como sequência para o laço.
- Não existe labels e goto em Python. Labels para escapar em conjunto com break foram sugeridos em 2007 (PEP 3136) porém o mesmo não foi aceito.

Controle de Fluxo

- **While**: O bloco de código dentro do laço **while** é repetido enquanto a condição do laço estiver sendo avaliada como verdadeira.

```
while <condição>:
    <bloco de código>
    continue
    break
else:
    <bloco de código>
```

Laço While

Tipos

- Tipos simples de dados pré-definidos no Python são:
  - Números (inteiros, reais, complexos, etc).
  - Texto (strings).

- Tipos compostos que funcionam como coleções, sendo os principais:
  - Lista.
  - Tupla.
  - Dicionário.

Tipos

- Os tipos no Python podem ser:
  - Mutáveis: permitem que os conteúdos das variáveis sejam alterados.
  - Imutáveis: não permitem que os conteúdos das variáveis sejam alterados.
- Em Python, os nomes de variáveis são referências, que podem ser alteradas em tempos de execução.
- Além disso, Python é uma linguagem Case Sensitive.
- Não é possível declarar constantes de tipos mutáveis.

Tipos

- **Números**:
  - Inteiro (int): i = 1
  - Real de ponto flutuante (float): f = 3.14
  - Complexo (complex): c = 3 + 4j

- Além dos números inteiros simples (4 bytes), existem também os inteiros longos, que tem dimensão arbitrária e são limitados pela memória disponível. As conversões entre inteiro e longo são realizadas de forma automática.

Tipos

- O Python tem uma série de operadores definidos para manipular números:
  - Operações aritméticas:
    - Soma (+).
    - Diferença (-).
    - Multiplicação (*).
    - Divisão (/): entre dois inteiros funciona igual à divisão inteira. Em outros casos, o resultado é real.

Tipos

- Divisão inteira (//): o resultado é truncado para o inteiro imediatamente inferior, mesmo quando aplicado em números reais, porém neste caso o resultado será real também.
- Módulo (%): retorna o resto da divisão.
- Potência (**): pode ser usada para calcular a raiz, através de expoentes fracionários (exemplo: 100 ** 0.5).
- Positivo (+).
- Negativo (-).

Tipos

- Operações lógicas:
  - Menor (<).
  - Maior (>).
  - Menor ou igual (<=).
  - Maior ou igual (>=).
  - Igual (==).
  - Diferente (!=).

Tipos

- **Texto**: São imutáveis, não é possível adicionar, remover ou mesmo modificar algum caractere de uma *string*. Para realizar essas operações, o Python precisa criar um nova *string*.
  - Tipos:
    - *String* padrão: s = 'Led Zeppelin'
    - *String unicode*: u = u'Björk'
  - A *string* padrão pode ser convertida para *unicode* através da função *unicode()*.

Tipos

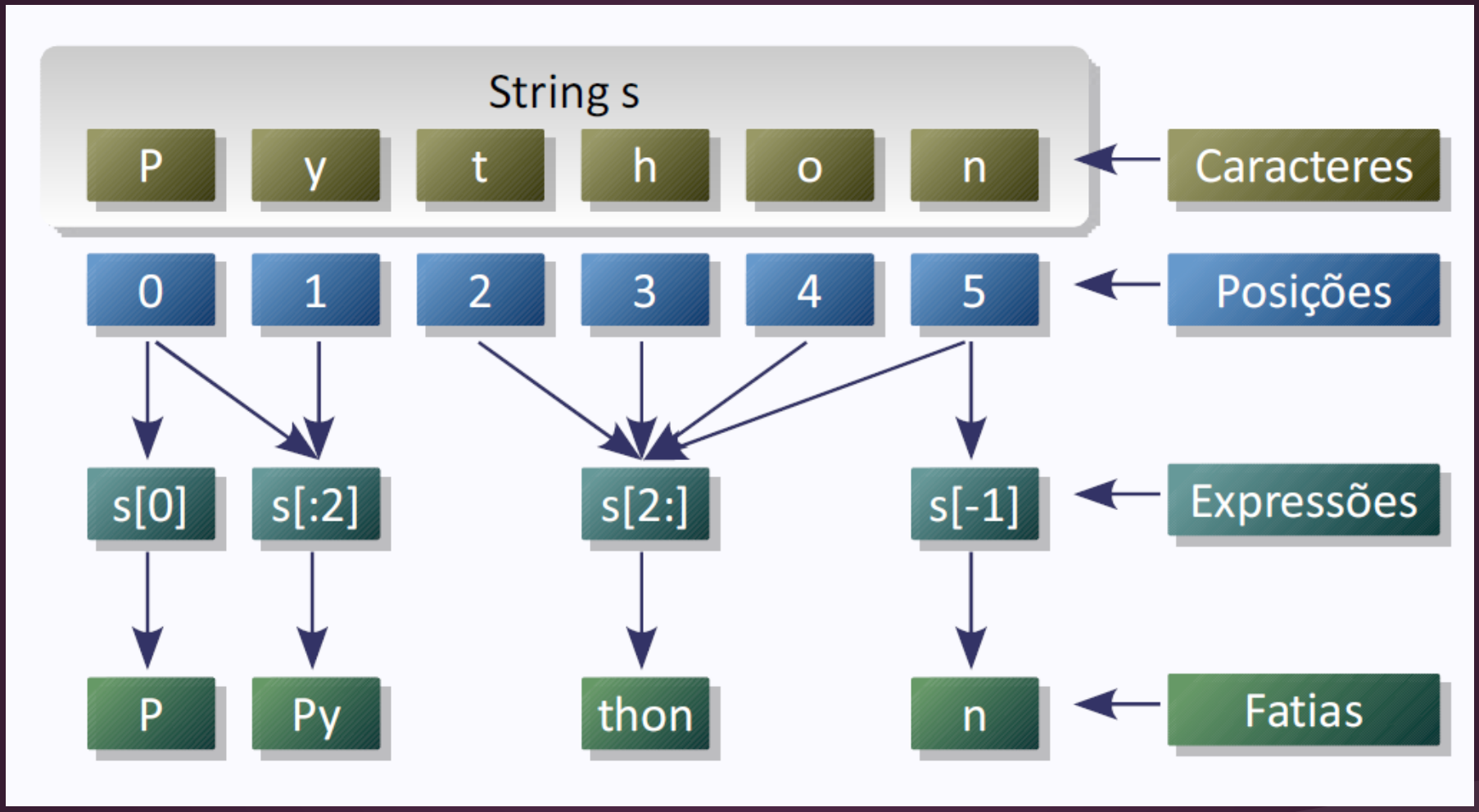
- A inicialização de strings pode ser:
  - Com aspas simples ou duplas.
  - Em várias linhas consecutivas, desde que seja entre três aspas simples ou duplas.
  - Sem expansão de caracteres (exemplo: s = r'\n', em que s conterá os caracteres “\” e “n”).

```
# -*- coding: latin1 -*-

s = 'Camel'

# Concatenação
print 'The ' + s + ' run away!'

# Interpolação
print 'tamanho de %s => %d' % (s, len(s))

# String tratada como sequência
for ch in s: print ch

# Strings são objetos
if s.startswith('C'): print s.upper()

# o que acontecerá?
print 3 * s
# 3 * s é consistente com s + s + s
```

Operações com String

Tipos

- Fatias (*slices*) de strings podem ser obtidas colocando índices entre colchetes após a *string*.



Fatiando Strings

Tipos

- Os índices no Python:
  - Começam em zero.
  - Contam a partir do fim se forem negativos.
  - Podem ser definidos como trechos, na forma [inicio:fim + 1:intervalo]. Se não for definido o inicio, será considerado como zero. Se não for definido o fim + 1, será considerado o tamanho do objeto. O intervalo (entre os caracteres), se não for definido, será 1.

Tipos

- Várias outras funções para tratar texto estão implementadas no módulo *string* e podem ser importadas para otimizar seu uso.

Tipos

- **Listas:** são coleções heterogêneas de objetos, que podem ser de qualquer tipo, inclusive outras listas.

- As listas no Python são mutáveis, podendo ser alteradas a qualquer momento. Listas podem ser fatiadas da mesma forma que as strings, mas como as listas são mutáveis, é possível fazer atribuições a itens da lista.

```
# Uma nova lista: Brit Progs dos anos 70
progs = ['Yes', 'Genesis', 'Pink Floyd', 'ELP']

# Varrendo a lista inteira
for prog in progs:
    print prog

# Trocando o último elemento
progs[-1] = 'King Crimson'

# Incluindo
progs.append('Camel')

# Removendo
progs.remove('Pink Floyd')

# Ordena a lista
progs.sort()

# Inverte a lista
progs.reverse()
```

Manipulando Listas

Tipos

- **Tuplas**: Semelhantes as listas, porém são imutáveis, ou seja, não se pode acrescentar, apagar ou fazer atribuições aos itens.
  - Sintaxe: tupla = (a, b, ..., z)
    - Os parênteses são opcionais.
- Os elementos de uma tupla podem ser referenciados da mesma forma que os elementos de uma lista:
  - primeiro_elemento = tupla[0]

Tipos

- Listas podem ser convertidas em tuplas:
  - tupla = tuple(lista)

- E tuplas podem ser convertidas em listas:
  - lista = list(tupla)

- Embora a tupla possa conter elementos mutáveis, esses elementos não podem sofrer atribuição, pois isto modificaria a referência ao objeto, além disso as tuplas são mais eficientes do que as listas convencionais, pois consomem menos recursos computacionais.

Tipos

- **Outros tipos de sequências**:
  - *set*: sequência mutável unívoca (sem repetições) não ordenada.
  - *frozenset*: sequência imutável unívoca não ordenada.

- Os dois tipos implementam operações de conjuntos, tais como: união, interseção e diferença.

```
# Conjuntos de dados
s1 = set(range(3))
s2 = set(range(10, 7, -1))
s3 = set(range(2, 10, 2))

# Exibe os dados
print 's1:', s1, '\ns2:', s2, '\ns3:', s3

# União
s1s2 = s1.union(s2)
print 'União de s1 e s2:', s1s2

# Diferença
print 'Diferença com s3:', s1s2.difference(s3)

# Interseção
print 'Interseção com s3:', s1s2.intersection(s3)
```

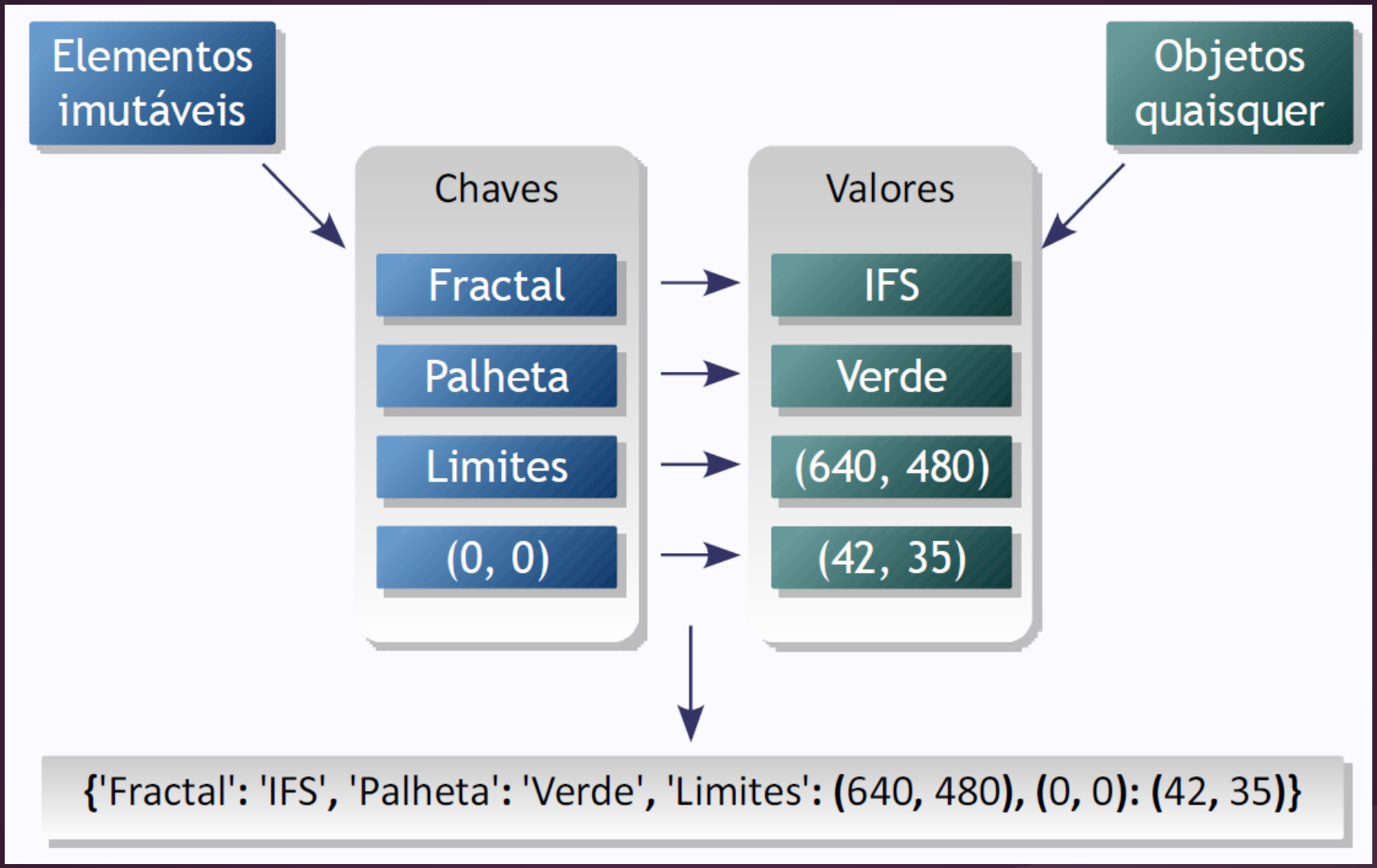
Manipulando Sets

Tipos

- **Dicionários**: Um dicionário é uma lista de associações compostas por uma chave única e estruturas correspondentes. Dicionários são mutáveis, tais como as listas.

- A chave precisa ser de um tipo imutável, geralmente são usadas strings, mas também podem ser tuplas ou tipos numéricos. Já os itens dos dicionários podem ser tanto mutáveis quanto imutáveis.

- O dicionário do Python não fornece garantia de que as chaves estarão ordenadas.



Estrutura de um Dicionario

Tipos

- **Booleano**: o tipo booleano (*bool*) é uma especialização do tipo inteiro (*int*). O verdadeiro é chamado *True* e é igual a 1, enquanto o falso é chamado *False* e é igual a zero.

- Os seguintes valores são considerados falsos:
  - False (falso).
  - None (nulo).
  - 0 (zero).
  - ' ' (string vazia).
  - [] (lista vazia).
  - () (tupla vazia).
  - {} (dicionário vazio).

Tipos

- São considerados verdadeiros todos os outros objetos fora da lista acima.

Cópia

- Como em python tudo é referência, ao fazermos uma atribuição apenas copiamos a referência
- Para copiar o objeto usamos o módulo *copy*
  - copy.copy(x)
    - "cópia rasa"
  - copy.deepcopy(x)
    - "cópia profunda"

Funções

- **Funções**: são blocos de código identificados por um nome, que podem receber parâmetros pré-determinados.

```
def func(parametro1, parametro2=padrao):
    """Doc String
    """
    <bloco de código>
    return valor
```

Funções

- No Python, as funções:
  - Podem retornar ou não objetos.
  - Aceitam *Doc Strings*.
  - Aceitam parâmetros opcionais (com *defaults*). Se não for passado o parâmetro será igual ao *default* definido na função.
  - Aceitam que os parâmetros sejam passados com nome. Neste caso, a ordem em que os parâmetros foram passados não importa.
  - Tem *namespace* próprio (escopo local), e por isso podem ofuscar definições de escopo global.

Funções

- *Doc Strings* são *strings* que estão associadas a uma estrutura do Python. Nas funções, as *Doc Strings* são colocadas dentro do corpo da função, geralmente no começo.

- O objetivo das *Doc Strings* é servir de documentação para aquela estrutura.

```
def fib(n):
    """Fibonacci:
    fib(n) = fib(n - 1) + fib(n - 2) se n > 1
    fib(n) = 1 se n <= 1
    """

    # Dois primeiros valores
    l = [1, 1]

    # Calculando os outros
    for i in range(2, n + 1):
        l.append(l[i -1] + l[i - 2])

    return l
```

Exemplo de uma Função em Python

Funções

- A passagem de parâmetros simples é feita por cópia.
- Deve-se usar global para alterar a variável deste forma:

```
1  n = 0
2
3  def test():
4      global n
5      n = 1
6
7  # before = 0
8  print 'before = ', n
9  test()
10 # after = 1
11 print 'after = ', n
```

Módulos

- **Módulos**: são arquivos fonte que podem ser importados para um programa. Podem conter qualquer estrutura do Python e são executados quando importados.

- Eles são compilados quando importados pela primeira vez e armazenados em arquivo (com extensão ".pyc" ou ".pyo"), possuem *namespace* próprio e aceitam *Doc Strings*.

Módulos

- Os módulos são carregados através da instrução *import*. Desta forma, ao usar alguma estrutura do módulo, é necessário identificar o módulo. Isto é chamado de importação absoluta.

```
import os
print os.name
```

Módulos

- Também possível importar módulos de forma relativa:

```
from os import name
print name
```

- O caractere "*" pode ser usado para importar tudo que está definido no módulo:

```
from os import *
print name
```

Módulos

- O módulo principal de um programa tem a variável *__name__* igual à "*__main__*", então é possível testar se o módulo é o principal usando:

```
if __name__ == "__main__":
    # Aqui o código só será executado
    # se este for o módulo principal
    # e não quando ele for importado por outro programa
```

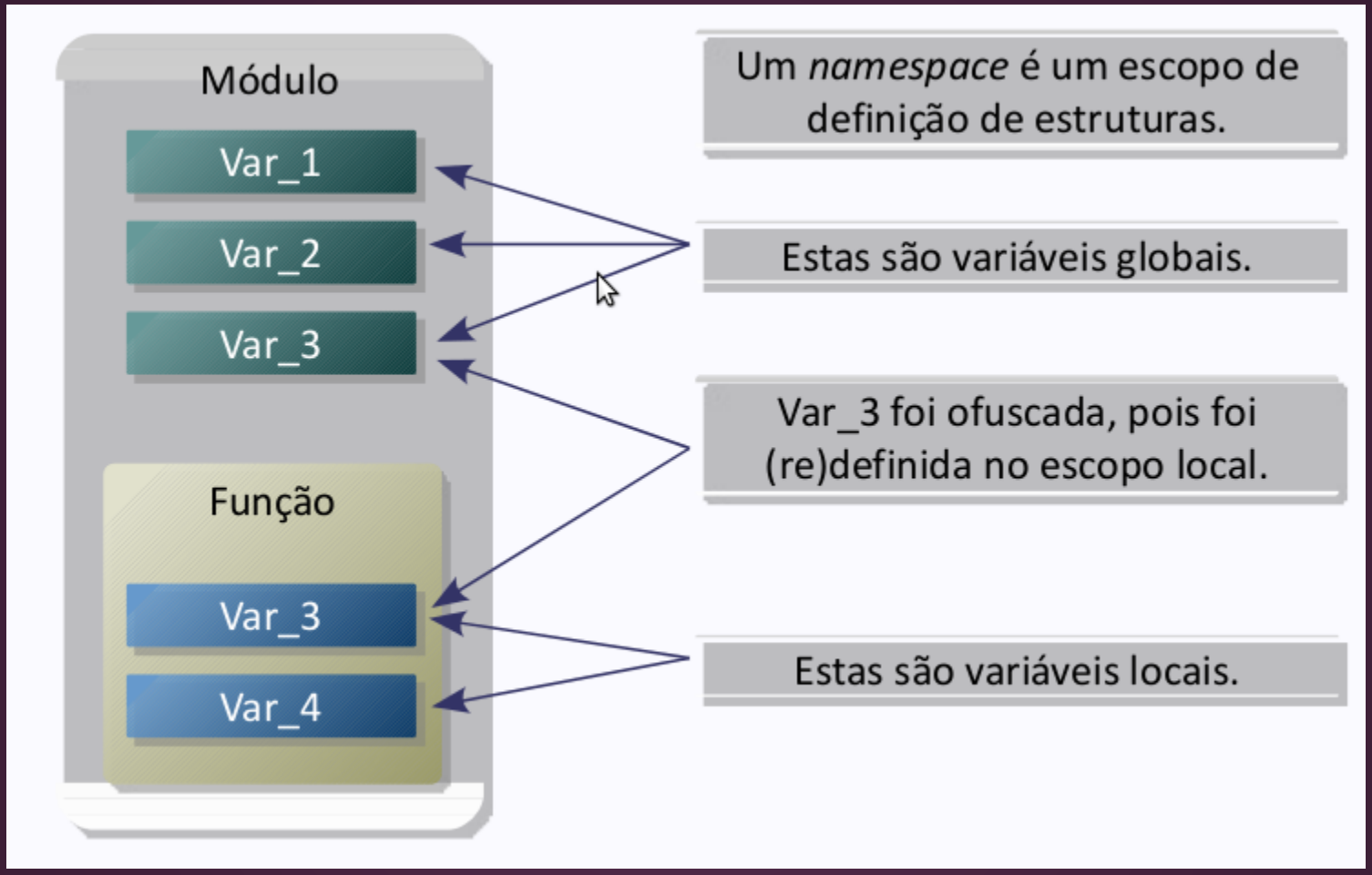
Módulos

- A integração com outras linguagens de programação como C (Cython) e Java (Jython), além do fato de que é possível chamar e retornar programas utilizando Python tornam a mesma uma boa linguagem para integrar módulos escritos em diferentes linguagem em um mesmo projeto.

Escopo de Nomes

- **Escopo de nomes**: é mantido através de *Namespaces*, que são dicionários que relacionam os nomes dos objetos (referências) e os objetos em si.

- Normalmente, os nomes estão definidos em dois dicionários, que podem ser consultados através das funções *locals()* e *globals()*. Estes dicionários são atualizados dinamicamente em tempo de execução.



Estrutura do Escopo de Nomes

Escopo de Nomes

- Variáveis globais podem ser ofuscadas por variáveis locais (pois o escopo local é consultado antes do escopo global). Para evitar isso, é preciso declarar a variável como global no escopo local.

Bibliotecas

- Python vem com “baterias inclusas”, em referência a vasta biblioteca de módulos e pacotes que é distribuída com o interpretador.

- Alguns módulos importantes da biblioteca padrão:
  - **Matemática**: *math, cmath, decimal e random*.
  - **Sistema**: *os, glob, shutils e subprocess*.
  - **Threads**: *threading*.
  - **Persistência**: *pickle e cPickle*.
  - **XML**: *xml.dom, xml.sax e elementTree*.
  - **Configuração**: *ConfigParser e optparse*.
  - **Tempo**: *time e datetime*.

Bibliotecas

- Acesso direto à memória só é possível com o uso da biblioteca ctypes, que trabalha de forma a possibilitar o uso de tipos compatíveis com C, bem como bibliotecas compartilhadas e DLLs do mesmo.

Matemática

- O módulo *math* define funções logarítmicas, de exponenciação, trigonométricas, hiperbólicas e conversões angulares, entre outras. Já o módulo *cmath*, implementa funções similares, porém feitas para processar números complexos.

```python
#!/usr/bin/env python
# *-* coding: utf-8 *-*
import math
a = 9
b = math.sqrt(a)
print(b) # Imprime 3.0
```

```python
#!/usr/bin/env python
# *-* coding: utf-8 *-*
a = 2 ** 23
```

Exemplos do uso da Biblioteca math

Arquivos

- Os arquivos no Python são representados por objetos do tipo *file*, que oferecem métodos para diversas operações de arquivos.

- Arquivos podem ser abertos para leitura ('r', que é o *default*), gravação ('w') ou adição ('a'), em modo texto ou binário ('b').

Arquivos

- Escrever:
  - f = open('/home/igor/Python/texto','w')
  - f.write('Ola Mundo!')

- Como o modo padrão é texto, não é necessário a extensão .txt. Para fechar o arquivo, fazemos:
  - f.close()

Arquivos

- Existem várias maneiras de ler o que está escrito, sendo elas:
  - Método *read()* usado sem nenhum argumento, mostrará tudo que esta no aquivo.
  - Método *readline()* sem argumento, ira ler todo o conteúdo de uma linha.
  - Método *readlines()* retorna todo o coteúdo do arquivo numa lista. Cada linha fica em um índice da lista no formato de uma *string*.

Arquivos

- Uma observação importante sobre os métodos *read()* e *readline()* é a possibilidade de passar argumentos na chamado do método, isso possibilita que se quisermos ler apenas os três primeiros caracteres (*bytes*), podemos usar da seguinte maneira:

  - ler = f.read(3)

Arquivos

- A persistência é possível através do módulo pickle e cPickle(um pickle escrito em C, até 1000 vezes mais rápido);

Toggle line numbers

```
1 # Save a dictionary into a pickle file.
2 import pickle
3
4 favorite_color = { "lion": "yellow", "kitty": "red" }
5
6 pickle.dump( favorite_color, open( "save.p", "wb" ) )
```

Toggle line numbers

```
1 # Load the dictionary back from the pickle file.
2 import pickle
3
4 favorite_color = pickle.load( open( "save.p", "rb" ) )
5 # favorite_color is now { "lion": "yellow", "kitty": "red" }
```

Arquivos

- O pickle tem como função salvar um objeto em formato ‘byte stream’ ou retornar um arquivo byte stream novamente para um objeto.
- Não é seguro abrir um byte stream de origem desconhecida através deste programa, o mesmo pode conter um código malicioso.

Exceções

- **Exceções**: Quando ocorre uma falha no programa (como uma divisão por zero, por exemplo) em tempo de execução, uma exceção é gerada.

- Se a exceção não for tratada, ela será propagada através das chamadas de função até o módulo principal do programa, interrompendo a execução.

Exceções

- A instrução *try* permite o tratamento de exceções no Python. Se ocorrer uma exceção em um bloco marcado com *try*, é possível tratar a exceção através da instrução *except*. Podem existir vários blocos *except* para o mesmo bloco *try*.
- Se *except* recebe o nome da exceção, só esta será tratada. Se não for passada nenhuma exceção como parâmetro, todas serão tratadas.

```
try:
    print 1/0

except ZeroDivisionError:
    print 'Erro ao tentar dividir por zero.'
```

Exemplo de Tratamento de Exceção

Exceções

- O módulo *traceback* oferece funções para manipular as mensagens de erro.
- O tratamento de exceções pode possuir um bloco *else*, que será executado quando não ocorrer nenhuma exceção e um bloco *finally*, será executado de qualquer forma, tendo ocorrido uma exceção ou não.
- Novos tipos de exceções podem ser definidos através de herança a partir da classe *Exception*.

Exceções

- Hierarquia de Excessões:

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StandardError
      |    +-- BufferError
      |    +-- ArithmeticError
      |    |    +-- FloatingPointError
      |    |    +-- OverflowError
      |    |    +-- ZeroDivisionError
      |    +-- AssertionError
      |    +-- AttributeError
      |    +-- EnvironmentError
      |    |    +-- IOError
      |    |    +-- OSError
      |    |         +-- WindowsError (Windows)
      |    |         +-- VMSError (VMS)
      |    +-- EOFError
      |    +-- ImportError
      |    +-- LookupError
      |    |    +-- IndexError
      |    |    +-- KeyError
      |    +-- MemoryError
      |    +-- NameError
      |    |    +-- UnboundLocalError
      |    +-- ReferenceError
      |    +-- RuntimeError
      |    |    +-- NotImplementedError
      |    +-- SyntaxError
      |    |    +-- IndentationError
      |    |         +-- TabError
      |    +-- SystemError
      |    +-- TypeError
      |    +-- ValueError
      |         +-- UnicodeError
      |              +-- UnicodeDecodeError
      |              +-- UnicodeEncodeError
      |              +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
```

Introspecção

- Introspecção ou reflexão é capacidade do software de identificar e relatar suas próprias estruturas internas, tais como tipos, escopo de variáveis, métodos e atributos.

Função	Retorno
type(objeto)	O tipo (classe) do objeto
id(objeto)	O identificador do objeto
locals()	O dicionário de variáveis locais
globals()	O dicionário de variáveis globais
vars(objeto)	O dicionário de símbolos do objeto
len(objeto)	O tamanho do objeto
dir(objeto)	A lista de estruturas do objeto
help(objeto)	As *Doc Strings* do objeto
repr(objeto)	A representação do objeto
isinstance(objeto, classe)	Verdadeiro se objeto deriva da classe
issubclass(subclasse, classe)	Verdadeiro se subclasse herda classe

Funções nativas do Interpretador para Introspecção

Classes

- No Python, novos objetos são criados a partir das classes através de atribuição. O objeto é uma instância da classe, que possui características próprias.
- Quando um novo objeto é criado, o construtor da classe é executado. Em Python, o construtor é um método especial, chamado *__new__()*. Após a chamada ao construtor, o método *__init__()* é chamado para inicializar a nova instância.

Classes

- Um objeto continua existindo na memória enquanto existir pelo menos uma referência a ele. O interpretador Python possui um recurso chamado coletor de lixo (*Garbage Collector*) que limpa da memória objetos sem referências.

Classes

- Em Python:
  - Quase tudo é objeto, mesmo os tipos básicos, como números inteiros.
  - Tipos e classes são unificados.
  - Os operadores são na verdade chamadas para métodos especiais.
  - As classes são abertas.

```
class Classe(supcl1, supcl2):
    """
    Isto é uma classe
    """
    clsvar = []

    def __init__(self, args):
        """
        Inicializador da classe
        """
        <bloco de código>

    def __done__(self):
        """
        Destrutor da classe
        """
        <bloco de código>

    def metodo(self, params):
        """
        Método de objeto
        """
        <bloco de código>
```

Sintaxe de uma Classe

Classes

- Métodos de objeto podem usar atributos e outros métodos do objeto. A variável *self*, que representa o objeto também precisa ser passado de forma explícita.
- Métodos de classe podem usar apenas atributos e outros métodos de classe.
- Métodos estáticos são aqueles que não tem ligação com atributos do objeto ou da classe. Funcionam como as funções comuns.

Classes

- As variáveis e os métodos são escritos precedidos pelo nome da classe e por um ponto (.):
  - Assim, uma variavel *var* definida numa classe *C* é escrita *C.var*.
- Uma nova instância da classe é criada usando *"nome_da_Classe"().*

Classes

- Em Python, não existem variáveis e métodos privados (que só podem ser acessados a partir do próprio objeto). Ao invés disso, é usada uma convenção, usar um nome que comece com sublinhado (_).

```
class A:
   a = 1 # atributo publico

class B(A): # classe B está herdando A
   _c = 3 # atributo considerado privado

   def __init__(self): # método especial que pode ser considerado "construtor"
     print self.a
     print self._c

a = A()
print isinstance(a, B) # ''Objeto a'' é uma instância da ''classe B''? Falso.

a = B() # Instancía o ''objeto a'' na ''classe B'' e imprime os atributos da classe.
print isinstance(a, B) # ''Objeto a'' é uma instância da ''classe B''?Verdadeiro.
```

Exemplo de uma Classe

Classes Abertas

- **Classes abertas**: No Python, as classes que não são *builtins* podem ser alteradas em tempo de execução, devido a natureza dinâmica da linguagem.

- É possível acrescentar métodos e atributos novos, por exemplo. A mesma lógica se aplica aos objetos.
- De mesmo modo isto pode funcionar como 'templates' e classes abstratas.

Herança Simples

- A forma comum de herança é chamada de herança simples, na qual a nova classe é derivada de apenas uma classe já existente, porém é possível criar várias classes derivadas, criando uma hierarquia de classes.
- Para localizar os métodos e atributos, a hierarquia é seguida de baixo para cima, de forma similar a busca nos *namespaces* local e global.

Herança Simples

- A nova classe pode implementar novos métodos e atributos e herdar métodos e atributos da classe antiga (que também pode ter herdado de classes anteriores), porém estes métodos e atributos podem substituídos na nova classe.

```
class Pendrive(object):

    def __init__(self, tamanho, interface='2.0'):

        self.tamanho = tamanho
        self.interface = interface


class MP3Player(Pendrive):

    def __init__(self, tamanho, interface='2.0', turner=False):

        self.turner = turner
        Pendrive.__init__(self, tamanho, interface)


mp3 = MP3Player(1024)

print '%s\n%s\n%s' % (mp3.tamanho, mp3.interface, mp3.turner)
```

A classe *MP3Player* é derivada da classe *Pendrive*.

Exemplo de Herança Simples

Herança Múltipla

- Na herança múltipla, a nova classe deriva de duas ou mais classes já existentes.

```
class Azul():
    cor = 'azul'
    nivel = 1
    tom = 'marinho'
    def ehVermelho(self):
        return False

class Vermelho:
    cor = 'vermelho'
    nivel = 2

    def ehVermelho(self):
        return True

class Minha():
    def printcor(self):
        print self.cor
        print self.nivel
        print self.tom

class MinhaCor(Minha, Vermelho, Azul):
    def ehVermelho(self):
        return "Sempre!"


mc = MinhaCor()
mc.printcor()
print mc.ehVermelho()
print dir(mc)
```

Exemplo de Herança Multipla

Herança Múltipla

- Porém tem que ficar atento a ordem da herança, pois Python importa da esquerda para a direita.
- Como no exemplo, temos alguns atributos repetidos, como a “cor”,”nível” e “tom”, e quando executamos o código, o que será impresso são os valores herdados da classe Vermelho, sendo assim os atributos “cor” e “nível” do Azul não se sobrepõem aos já herdados.

Sobrecarga de Operadores

- No Python, o comportamento dos operadores é definido por métodos especiais, porém tais métodos só podem ser alterados nas classes abertas.
- Por convenção, os métodos especiais têm nomes que começam e terminam com "__".

Operador	Método	Operação
+	__add__	adição
-	__sub__	subtração
*	__mul__	multiplicação
/	__div__	divisão
//	__floordiv__	divisão inteira
%	__mod__	módulo
**	__pow__	potência
+	__pos__	positivo
-	__neg__	negativo
<	__lt__	menor que
>	__gt__	maior que
<=	__le__	menor ou igual a
>=	__ge__	maior ou igual a
==	__eq__	Igual a

Lista de Operadores e Métodos Correspondentes

Coleções

- Em Python existem métodos especiais para lidar com objetos que funcionam como coleções (da mesma forma que as listas e os dicionários), possibilitando o acesso aos itens que fazem parte da coleção.

```
class Mat(object):
    """
    Matriz esparsa
    """

    def __init__(self):
        """
        Inicia a matriz
        """

        self.itens = []
        self.default = 0

    def __getitem__(self, xy):
        """
        Retorna o item para X e Y ou default caso contrário
        """

        i = self.index(xy)
        if i is None:
            return self.default
        return self.itens[i][-1]


    def __setitem__(self, xy, data=0):
        """
        Cria novo item na matriz
        """

        i = self.index(xy)
        if not i is None:
            self.itens.pop(i)
        self.itens.append((xy, data))

    def __delitem__(self, xy):
        """
        Remove um item da matriz
        """

        i = self.index(xy)
        if i is None:
            return self.default
        return self.itens.pop(i)
```

Exemplo de Coleção

```
  def dim(self):
      """
      Retorna as dimensões atuais da matriz
      """
      x = y = 0
      for xy, data in self.itens:
          if xy[0] > x: x = xy[0]
          if xy[1] > y: y = xy[1]

      return x, y

def __repr__(self):
    """
    Retorna uma representação do objeto como texto
    """

    r = 'Dim: %s\n' % repr(self.dim())
    X, Y = self.dim()

    for x in xrange(1, X + 1):
        for y in xrange(1, Y + 1):
            r += ' %s = %3.1f' % (repr((x, y)),
              float(self.__getitem__((x, y))))
     r += '\n'
return r
```

Exemplo de Coleção

Concorrência

- É possível através das bibliotecas ‘multiprocessing’ e ‘threading’.
- Em CPython, um mutex (mutual exclusion) chamado GIL (Global Interpreter Lock) é necessário para impedir mais de uma thread de executar o mesmo trecho de código ao mesmo tempo pois a memória compartilhada do mesmo não é thread-safe.

Concorrência

- Jython e IronPython não utizam GIL e por este motivo podem aproveitar um ambiente com multiplos processadores.
- Stackless Python é uma implementação de Python que não faz uso da pilha de C, desta forma podendo aceitar diversas threads rodando em paralelo, porém sua implementação ainda é voltada a um único núcleo.

Avaliação da LP

- Alta legibilidade:
  - Projeto da LP foi desenvolvido com enfoque na legibilidade.
  - Delimitação por identação.
- Redigibilidade:
  - Bibliotecas padrão bastante abrangente

Avaliação da LP

- Confiabilidade:
  - Tratamento de exceções.
  - Variáveis podem armazenar dados de qualquer tipo.
- Eficiência:
  - Verificação dinâmica de tipos.
  - Coletor de lixo.

Avaliação da LP

- Facilidade de Aprendizado.
- Reusabilidade.
- Portabilidade.

Características de Projeto

- Python foi desenvolvida com a filosofia de enfatizar a importância do esforço do programador sobre o esforço do computador.
- Prioriza legibilidade de código sobre velocidade ou expressividade.

Características de Projeto

- The Zen of Python

The Zen of Python

```
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!
```

Mais informações

- Documentação e tutoriais:

http://python.org/doc/

- Websites

http://www.learnpython.org/