Allison Machado - allisonline.net

Humberto Rocha - humberto.io

Linguagem de Programação Python

Brasília - DF, Brasil

11 de agosto de 2013

Resumo

O presente trabalho tem como objetivo apresentar a linguagem de programação Python, uma linguagem elegante que possui uma biblioteca padrão ampla capaz de prover suporte a muitas tarefas de programação comuns, como processamento de texto, tratamento de arquivos, atividades de rede e etc. A linguagem será apresentada por meio de uma perspectiva científica, analítica e bem resumida. Será apresentada a história da linguagem, tópicos sobre a análise léxica, estudo da estrutura da linguagem e exemplos de utilização da linguagem de forma prática.

Palavras-chaves: Python, programação, processamento, variáveis, linguagens, computação.

Abstract

This paper aims to present the Python programming language, an elegant language that has a wide standard library able to provide support for many common programming tasks such as text processing, file handling, networking activities and so on. The language will be presented through an analytical, well summarized and scientific perspective. It will be presented the history of the language, topics about lexical analysis, a language structure study and examples of practical language usage.

Keywords: Python, programming, processing, variables, languages, computing.

Lista de ilustrações

Lista de abreviaturas e siglas

BNF	*Backus–Naur Form*;
ASCII	*American Standard Code for Information Interchange*;
LF	*Line Feed*;
CR	*Carriage Return*;
ISO	*International Organization for Standardization.*

Sumário

1 História

Python é uma linguagem de programação de alto nível criada por Guido van Rossum, da qual começou seu desenvolvimento em dezembro de 1989, influenciado pelo seu trabalho na linguagem ABC e buscando preencher o espaço entre a complexidade do C, os problemas do shell script.

1.1 A Influência do ABC

No início dos anos 80, Guido trabalhou no time de desenvolvimento da linguagem ABC no Instituto de Pesquisa Nacional para Matemática e Ciência da Computação (CWI) em Amsterdam na Holanda, onde teve a oportunidade de entrar em contato com uma filosofia de design de linguagem que futuramente veio ser o pilar principal para o desenvolvimento do Python.

ABC era uma linguagem com foco em ser fácil de ensinar, aprender e usar (ROSSUM, 2011). Ela buscava reutilizar conhecimentos já aprendidos e dominados (como o inglês e a matemática) e os unir da forma mais natural possível afim de reduzir a resistência das pessoas ao aprender programação.

Ela tinha como público alvo os cientistas e pesquisadores que não eram programadores, mas precisavam da programação em seu dia a dia.

Segundo Rossum (2011), devido a diversos fatores a linguagem ABC não alcançou sucesso, dentre eles os principais foram:

- Não havia uma comunidade formada para acolher a linguagem;
- O público alvo da linguagem não possuía fácil acesso ao hardware para aprender, e os que tinham já eram usuários avançados preferindo programar na linguagem que estavam acostumados;
- Não havia internet, o que tornava extremamente difícil espalhar uma nova linguagem;
- A linguagem era monolítica e não tinha sido pensada para prover extensibilidade e integração com o sistema, o que tornava o uso de algo não coberto pelos desenvolvedores algo extremamente complexo.

1.2 O Surgimento da linguagem

Em 1986 Guido engajou em um projeto diferente no CWI, era um projeto de desenvolvimento para um sistema operacional distribuído chamado de Amoeba.

Logo surgiu a necessidade de se desenvolver uma linguagem de script para o sistema, então, devido a sua liberdade e conhecimento prévio na área, Guido começou a desenvolver em dezembro de 1989 como um mini projeto uma linguagem que se inspirava nos pontos fortes da ABC como o design simples, a ideia de tipos de dados de alto nível (tupla, lista, dicionários) e o agrupamento de bloco por indentação, deixando para trás o que era mais problemático na linguagem como o uso de palavras chaves maiúsculas, o *desing* monolítico (não extensível) e a falta de integração com o sistema.

Outra linguagem que influenciou Guido na criação de sua linguagem foi Modula-3, da qual pegou emprestado o tratamento de exceções e a ideia de módulos.

Em 1990 Python ficou famoso dentro do CWI e no dia 20 de fevereiro de 1991 foi lançado sua primeira versão (0.9.0) como Open Source para o público geral.

Em 2001 a Python Software Foundation foi formada, uma organização sem fins lucrativos criada para garantir os interesses da linguagem e da comunidade que cresceu com a linguagem.

Atualmente Python possui duas versões que seguem em paralelo devido a melhorias na linguagem que quebram com a retrocompatibilidade. Estas versões são a 3.x que é a mais atual e a 2.x que seguirá sendo atualizada até que a maioria dos pacotes da linguagem tenham sido devidamente portados para a 3.x..

1.3 O nome Python

Apesar de estar no logo o nome python não se refere a uma espécie de cobra.

O nome Python é uma homenagem a uma série inglesa de comédia *Monty Python's Flying Circus*.

2 Python - Tópicos sobre a Análise Léxica

O objetivo deste capítulo é expor ideias léxicas importantes, conduzindo o leitor a familiarização com a linguagem.

Atualmente existe apenas uma implementação da linguagem que é mundialmente difundida e usada, porém é importante citar que existem implementações alternativas para uso e aplicações particulares. A descrição léxica e sintática oficial da linguagem usa uma modificação da notação BNF (PYTHON SOFTWARE FOUNDATION, 2013a), exemplificada a seguir.

```
name      ::=  lc_letter (lc_letter | "_")*
lc_letter ::=  "a"..."z"
```

Cada regra começa com um nome (que será definido na regra) e um (::=), indicando que a seguir virá a definição do nome. Uma barra vertical (|) indica alternativas. Um (*) indica zero ou mais ocorrências do item anterior, similarmente um (+) indica uma ou mais repetições. Algo que vem entre colchetes ([ ]) é opcional. Os parênteses são usados apenas para agrupamento e strings literais são indicadas entre aspas. Os três pontos entre os caracteres acima indicam que podem ser usados quaisquer caracteres na lista de a até z.

2.1 Organização

Um programa Python é dividido em um certo número de linhas lógicas. As linhas lógicas podem ser formadas por uma ou mais linhas físicas. Uma linha física é uma sequência de caracteres terminado por uma sequência de fim de linha. Em arquivos de código fonte, qualquer das sequências conhecidas podem ser usadas - a forma Unix utilizando LF (line feed), o Windows ASCII usando a sequência de CR LF (return followed by linefeed), ou a forma Macintosh usando o CR (return). Todas estas formas podem ser usadas, independentemente da plataforma.

Um comentário começa com um jogo da velha (#) e termina no fim da linha física. Os comentários são ignorados pela sintaxe. Duas ou mais linhas físicas podem ser unidas em linhas lógicas usando caracteres de barra invertida (\). Quando uma linha física termina em uma barra que não faz parte de uma string literal ou comentário, ela se junta com a seguinte, formando uma única linha. Isso é muito usado para manter o código mais legível. Por exemplo:

```
if 1900 < year < 2100 and 1 <= month <= 12 \
    and 1 <= day <= 31 and 0 <= hour < 24 \
    and 0 <= minute < 60 and 0 <= second < 60:
        return 1
```

A linha lógica que contém apenas espaços, tabulações e, possivelmente, um comentário, é ignorada.

Os espaços em branco (espaços e *tabs*), no início de uma linha lógica são usados para determinar o agrupamento de declarações. Portanto não exitem chaves como em C para separar blocos agrupados de instruções, mas estes blocos são identificados pela indentação do texto. Esta característica contribui muito para a legibilidade e a limpeza dos códigos.

2.2 Tokens e Palavras Reservadas

Temos a seguintes categorias de tokens: identificadores, palavras-chave, literais (literais são valores constantes de algum tipo determinado, como um inteiro por exemplo), operadores, e delimitadores. Caracteres brancos, como espaços e tabs, não são tokens mas servem para separar tokens.

Identificadores (também conhecidos como nomes) são descritos pela seguinte definição léxica:

```
identifier ::=  (letter|"_") (letter | digit | "_")*
letter     ::=  lowercase | uppercase
lowercase  ::=  "a"..."z"
uppercase  ::=  "A"..."Z"
digit      ::=  "0"..."9"
```

Identificadores são de tamanho ilimitado e também são *case sensitive*.

A seguir é apresentada uma lista de palavras reservadas da linguagem, portanto não podem ser usadas como identificadores:

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

2.3 Codificação e Strings

É possível explicitar no arquivo fonte a codificação usada na escrita. Se em um comentário na primeira ou segunda linha satisfaz a seguinte expressão regular:

```
coding[=:]\s*([-\w.]+)
```

Fazendo com que este comentário seja processado e usado como uma declaração de codificação.

Por exemplo:

```
# -*- coding: <encoding-name> -*-
```

Strings literais são definidas pelas seguintes regras:

```
    stringliteral   ::=  [stringprefix](shortstring | longstring)
    stringprefix    ::=  "r" | "u" | "ur" | "R" | "U"
                         | "UR" | "Ur" | "uR"
    shortstring     ::=  ""shortstringitem* ""
                         | " shortstringitem* "
    longstring      ::=  ""longstringitem* ""
                         | " longstringitem* "
    shortstringitem ::=  shortstringchar | escapeseq
    longstringitem  ::=  longstringchar | escapeseq
    shortstringchar ::=  <any source character except "\"
                          or newline or the quote>
    longstringchar  ::=  <any source character except "\">
    escapeseq       ::=  "\" <any ASCII character>
```

Os caracteres são definidos de acordo com a codificação usada na definição das primeiras linhas do arquivo (comentário de definição), caso nenhuma definição seja dada a ASCII é usada como padrão.

Strings podem ser colocados em aspas simples (') ou aspas duplas ("). Elas também podem ser colocados em grupos de três aspas simples ou duplas, com a vantagem de poderem ser definidas em mais de uma linha física (conforme a definição mostrada no *frame* anterior). O caractere de barra invertida (\) é usado para escapar de caracteres que têm significado especial, como uma nova linha por exemplo.

Strings literais podem, opcionalmente, ser prefixados com a letra 'r' ou 'R', e são chamadas de 'raw strings' podendo usar regras diferentes para interpretar sequências de escape de barra invertida. Um prefixo de 'u' ou 'U' faz com que a sequência seja Unicode. Estas strings utilizam o conjunto de caracteres Unicode conforme definido pelo Consórcio Unicode e ISO 10646.

2.4 Literais Numéricos

Em Python existem quatro tipos de literais numéricos: inteiros simples, inteiros longos, números de ponto flutuante e números imaginários.

Inteiros simples e longos são definidos pelas seguintes regras:

```
longinteger    ::=  integer ("l" | "L")
integer        ::=  decimalinteger | octinteger
                    | hexinteger | bininteger
decimalinteger ::=  nonzerodigit digit* | "0"
octinteger     ::=  "0" ("o" | "O") octdigit+ | "0" octdigit+
hexinteger     ::=  "0" ("x" | "X") hexdigit+
bininteger     ::=  "0" ("b" | "B") bindigit+
nonzerodigit   ::=  "1"..."9"
octdigit       ::=  "0"..."7"
bindigit       ::=  "0" | "1"
hexdigit       ::=  digit | "a"..."f" | "A"..."F"
```

Literais inteiros que estão acima do maior inteiro simples representável (no caso 2147483647 ao utilizar aritmética de 32 bits) são considerados automaticamente inteiros longos. Não há nenhum limite para o comprimento literais inteiros para além do que pode ser armazenado em memória disponível. Os números de ponto flutuante são definidos segundo a regra as regras a seguir:

```
floatnumber   ::=  pointfloat | exponentfloat
pointfloat    ::=  [intpart] fraction | intpart "."
exponentfloat ::=  (intpart | pointfloat) exponent
intpart       ::=  digit+
fraction      ::=  "." digit+
exponent      ::=  ("e" | "E") ["+" | "-"] digit+
```

Literais imaginários são usados para criar números complexos, eles representam a parte imaginária do número complexo (Im(c)).

```
imagnumber ::=  (floatnumber | intpart) ("j" | "J")
```

Para criar um número complexo sem que sua parte real (Re(c)) seja nula, faça a adição de um número. Por exemplo, (3+4j).

3 Variáveis, Escopo e Vinculação

3.1 Memória

A gerência de memória em Python utiliza um heap que contém todos os objetos e estruturas de dados do programa. O controle do heap é feito pelo módulo Gerenciador de Memória (Python *memory manager*).

É importante compreender que a gestão do heap é realizado pelo Gerenciador e que o usuário não tem controle sobre ele, mesmo que manipule ponteiros de objetos para blocos de memória dentro desse heap. A alocação de espaço de heap para objetos Python e outros buffers internos é realizado sob demanda pelo gerenciador de memória Python - (PYTHON SOFTWARE FOUNDATION, 2013b).

Python aloca memória de forma transparente para o programador, gerencia os objetos usando um sistema de contagem de referência, e libera memória quando a contagem de referência de um objeto chega a zero (python possui coletor de lixo - *garbage collection*), ou seja, não há nenhuma variável referenciando determinado objeto.

3.2 Variáveis

Para entendermos a forma como as variáveis gerenciadas em Python, faremos uma comparação com a linguagem C e assim será possível construir o entendimento de como a gerência ocorre.

Quando uma declaração é feita em C, um bloco de memória é criado e será acessado por meio do nome da variável. Quando uma atribuição é feita a esta variável, o valor é colocado no endereço da variável declarada.

Figura 1 – int a = 1;

E portanto para cada variável criada, um novo endereço é reservado para ela. Se uma atribuição for feita, o novo valor irá sobrescrever o valor antigo no espaço de memória da variável.

Figura 2 – a = 2;

Atribuir uma variável a outra faz com que o valor atribuído seja armazenado em um novo endereço de memória.

Figura 3 – b = a;

Porém, em Python as variáveis se comportam mais como *tags* colocadas em endereços de memória. Quando uma atribuição é feita, uma tag com o nome da variável é colocada no endereço de memória do valor sendo *atribuido*.

Figura 4 – a = 1

Se o valor da variável for trocado, a tag com o nome da variável é apenas colocado em um novo endereço de memória contendo o novo valor. Não é necessário se preocupar com o valor antigo, pois como já foi dito, o *coletor de lixo* do Python cuidará da limpeza da memória - quando a contagem de referência de um objeto chegar a zero, a memória será liberada.

Figura 5 – a = 2

Atribuir uma variável a outra faz com que existam duas tags rotulando o mesmo endereço de memória - isto se chama *alias*. Veja a próxima figura, observe que existem duas tags rotulando o valor 2.

Figura 6 – b = a

No caso do exemplo acima (variáveis simples inteiras) quando uma nova atribuição for feita a qualquer das variáveis (a ou b) o rótulo apenas mudará para o novo valor.

Entretanto, comportamento dos *aliases* muda quando se trabalha com estruturas de dados mutáveis como listas - note que inteiros são tipos de dados não mutáveis em Python. Quando um *alias* ocorre com estes tipos de dados e uma instrução de atribuição é usado em uma das variáveis referentes ao *alias*, as duas variáveis terão o valor alterado. Observe:

```
# Cria uma nova lista
a = [1,2,3]

# Cria um alias
b = a
print id(a), id(b) # imprime o endereço das variáveis
                   # a e b que é o mesmo

# Adiciona o valor quatro em a e b
b.append(4)

print a,b # os valores impressos serão iguais - [1,2,3,4]
```

3.3 Escopo

Python tem apenas duas noções de escopo - global e local. Isto significa que se uma variável é declarada em uma função, ela é vinculado à função. Caso contrário, ela está vinculado ao estado global. Considere o seguinte programa:

```
aux = 10

def foo():
    aux = 5

foo() # aux global continua = 10
```

Após a execução da função foo(), a variável global aux não será alterada. A fim de modificar uma variável em âmbito global, estando fora deste escopo, você deve declará-la como global. A diferença relevante é explicada pelos seguintes trechos de código.

```
aux = 10

def foo():
    global aux
    aux = 5

foo() # aux = 5
```

Observe que variáveis definidas dentro de expressões (if/while/for/try/except) estão sempre no escopo global. Observe o exemplo:

```
if True:
    b = 21

print b # 21

if False:
    b = 2

print b # 21
```

Este exemplo se aplica a todos os outros tipos de expressões, não somente o if. Observe o contraste com linguagens como C, por exemplo, onde o escopo das variáveis estão contidas no escopo de bloco onde são declaradas. No código abaixo, em C, o segundo printf geraria um erro em tempo de compilação pois a variável aux estaria definida apenas dentro do bloco if.

```
int main()
{
    if(1) //True
    {
        int aux = 6;
        printf("%d\n", aux);
    }
    printf("%d\n", aux); // erro
    return 0;
}
```

Python é uma linguagem que usa escopo estático, onde a busca pela declaração de uma variável inicia-se pelas declarações locais. Caso não encontre, procura-se nas declarações do seu pai-estático (o pai-estático de um bloco é o bloco dentro do qual ele foi declarado). A busca continua até alcançar o bloco mais externo, caso não encontre, emite um erro de variável não declarada. Observe o exemplo no código a seguir:

```
def foo():
    aux = 5
    def goo():
        print aux # busca no pai
    goo()

foo()
```

O valor na saída será 5 pois a função goo() busca pela definição de aux na função foo() - seu pai-estático.

3.4 Vinculação

A vinculação de tipos no Python acontece de forma dinâmica, logo ocorre durante a execução de um programa e pode ser modificada durante a execução do programa. A declaração de variáveis no Python é feita de forma implícita. A primeira ocorrência de um nome de variável relacionado a uma atribuição em um programa constitui sua declaração implícita, portanto o interpretador Python sabe o tipo da variável por meio do tipo de dado que está sendo atribuído à variável no momento de sua declaração implícita.

Uma das vantagens desta forma de vinculação é a flexibilidade. Não é necessário se preocupar com declarações de tipos de dados no momento do desenvolvimento de um programa e de maneira geral o código fica mais limpo. Porém não há detecção de erros relacionados a tipos de dados antes da execução do código. Além disso, a implementação da linguagem é mais custosa. É necessário verificar tipos em tempo de execução mantendo descritores de tipos relacionados às variáveis do programa, e tratar a possibilidade de mudança de tamanho das variáveis.

4 A linguagem na Prática!

Nas próximas seções veremos exemplos do uso de variáveis, tipos de dados, strings, estruturas de dados ainda não mencionadas, iterações, tratamento de exceções, manipulação de arquivos e interoperabilidade com o Sistema Operacional. Isto por meio de scripts ou programas explicados.

4.1 Usando Variáveis

Nas listagens de código a seguir, quando o leitor observar uma sequência de três caracteres “>” (maior que) seguidos, poderá inferir que o código está sendo escrito diretamente no Interpretador Python - por meio de uma interface de linha de comando - de forma que o resultado de uma expressão é retornado imediatamente pelo interpretador e impresso na tela.

Conforme explicado, em python variáveis apontam para dados armazenados na memória (Ver ítem 3.2 - pág 12). Logo, na memória podemos ter valores de diferentes tipos como inteiros, booleanos, strings ou estruturas mais complexas como listas ou dicionários. No trecho de código a seguir, é definida uma variável chamada *port* e outra chamada *banner* que armazenam um inteiro e uma string respectivamente. Para concatenar as variáveis deve-se fazer uma conversão explícita (*type cast*) do tipo inteiro por meio da função str().

```
>>> port = 21
>>> banner = "FreeFloat FTP Server"
>>> print "[+] Checking for "+banner+"on port"+str(port)
[+] Checking for FreeFloat FTP Serveron port21
```

Python reserva espaço para as variáveis quando o programador as declara. Não é necessário declarar explicitamente o tipo das variáveis, o interpretador decide o tipo e a quantidade de espaço necessário para armazenar a variável de acordo com o valor da atribuição.

```
>>> port = 21
>>> type(port)
<type int>
>>> banner = "FreeFloat FTP Server"
>>> type(banner)
<type str>
```

4.2 Alguns Métodos de Strings

O módulo de strings contém uma grande variedade de funções para manipulação de strings e toda a documentação juntamente com a listagem dos métodos pode ser encontrada em (PYTHON SOFTWARE FOUNDATION, 2013d). Vamos examinar alguns métodos que são comumente utilizados. Estas funções são simples e fáceis de ler, garantindo a legibilidade do código. Note que se o processamento que está sendo realizado está ficando muito complexo (com declarações excessivas de **if - else** para lidar com casos especiais e funções de strings sem um propósito claro), a melhor saída pode ser o uso de expressões regulares - Python possui suporte a expressões regulares por meio do módulo **re** - ver (PYTHON SOFTWARE FOUNDATION, 2013c).

Considere os seguintes métodos: upper(), lower(), replace() e find(). Upper() converte os caracteres da string para maiúsculos e Lower() para minúsculo. Replace(old, new) substitui a ocorrência do parâmetro old pelo parâmetro new. Finalmente o método Find() indica o índice da primeira ocorrência da string passada como parâmetro. Veja os exemplos a seguir:

```
>>> banner = "FreeFloat FTP Server"
>>> print banner.upper()
FREEFLOAT FTP SERVER
>>> print banner.lower()
freefloat ftp server
>>> print banner.replace("FreeFloat","Ability")
Ability FTP Server
>>> print banner.find(FTP)
10
```

4.3 Listas

Python possui a estrutura de dados Lista capaz de armazenar arrays de tipos de dados. É possível construir listas de qualquer tipo de dado e existem métodos fornecidos pela linguagem que realizam operações fundamentais como, por exemplo, inserção, remoção, indexação, ordenação e outros. Observar os seguintes exemplos de utilização de listas.

```
>>> list1 = [physics, chemistry, 1997, 2000];
>>> list2 = [5,4,3,2,1,0]
>>> list1[0]
physics
>>> list2[1:5] # retorna do índice 0 até o 4
[4, 3, 2, 1]
>>> list2.sort() # ordena a lista
>>> list2 # valor da lista depois da ordenação
[0, 1, 2, 3, 4, 5]
>>> list1.index(physics) # retorna o índice de "physics"
0
>>> list2.append(-1) # insere o valor -1 no fim da lista
>>> list2 # valor da lista depois da inserção
[0, 1, 2, 3, 4, 5, -1]
```

4.4 Dicionários

A estrutura de dados *Dicionário* consiste de pares de itens - mapeamentos no formato chave e valor. Esta é uma generalização da ideia de acessar dados por índices, exceto que neste mapeamento os índices (ou chaves) podem ser de qualquer tipo imutável. Diferentemente das listas que são representadas por colchetes [], os dicionários são representados por chaves .

```
>>> lista = []
>>> type(lista)
<type list>
>>> dicionario = {}
>>> type(dicionario)
<type dict>
```

Na construção de um dicionário, cada chave é separada de seu valor por um caractere de dois pontos e os itens são separados entre si por vírgulas.

```
>>> services = {"ftp":21, "ssh":22, "smtp":25, "http":80}
>>> services.keys()
["ftp", "smtp", "ssh", "http"]
>>> services.items()
[("ftp", 21), ("smtp", 25), ("ssh", 22), ("http", 80)]
>>> services.has_key("ftp")
True
>>> services["ftp"]
21
```

O método .keys() acima retorna uma lista contendo todas as chaves existentes no dicionário e o método .items() retorna uma lista contendo os itens do dicionário.

4.5 Condicionais

A palavras reservadas **if**, **elif** e **else** avaliam expressões lógicas para determinar a sequência de execução do programa. Para exemplificar o uso destas construções, observe o script a seguir que é um exemplo de loop de eventos usado para a codificação de jogos - Esta interface de programação orientada a eventos é fornecida por uma biblioteca chamada Pygame - (PYGAME, 2013).

```
1  while running:
2      # event queue iteration
3      for event in pygame.event.get():
4          if event.type == QUIT:
5              running = False
6          elif event.type == KEYDOWN:
7              keydown_handler(event.key)
8          elif event.type == KEYUP:
9              keyup_handler(event.key)
10         else:
11             pass
```

No exemplo acima, o loop somente irá parar sua execução quando o usuário entrar com um sinal de saída (linha 4). Em cada iteração do loop, que neste caso ocorre para cada *frame* do jogo, os eventos serão avaliados e o jogo será atualizado. A biblioteca Pygame oferece suporte a detecção de vários tipos de eventos como, por exemplo, teclas pressionadas, posição e cliques do mouse, interações com joysticks, redimensionamento e tentativas de fechar janelas. Nas linhas 6 e 8 é verificado se alguma tecla foi pressionada e liberada, respectivamente, na detecção destes eventos o programa chama funções que devem tratar os eventos e interagir com o jogo da maneira correta (linhas 7 e 9) passando as teclas pressionadas para estas funções. Na linha 10 foi adicionado a cláusula **else** apenas para exemplificar seu uso, e dentro deste bloco foi declarado a sentença **pass** que significa uma operação nula - "nada acontece em sua execução".

4.6 Funções

Funções são blocos organizados de código reutilizável. Sendo possível atribuir um nome a um trecho de código, executá-lo aonde e quantas vezes for necessário. Python provê uma série de funções prontas para facilitar algumas tarefas do programador, porém é possível que o programador defina sua próprias funções. A palavra reservada **def** serve para isto.

```
>>> def greeting():
...     print "Hello World!"
...
>>> greeting()
Hello World!
```

É possível passar parâmetros para funções. Os parâmetros são utilizados por funções de diversas maneiras, podem ser processados, e alterar o comportamento das funções.

```
>>> def greeting(param):
...     print "Hello", param
...
>>> greeting("Humberto")
Hello Humberto
```

Para retornar algo de uma função, utilize a palavra reservada **return**.

```
>>> def greeting(param):
...     return "Hello " + param
...
>>> person = "Allison"
>>> print greeting(person)
Hello Allison
```

4.7 Iterações

Considere a seguinte situação - É necessário iterar sobre todos os ips da sub-rede 192.168.95.1/24 e verificar se estas máquinas estão com determinadas portas abertas escutando requisições. Usando um laço for para uma série de 1 a 255 é possível iterar sobre toda a sub-rede.

```
>>> for x in range(1,255):
...     print "192.168.95."+str(x)
...
192.168.95.1
192.168.95.2
192.168.95.3
192.168.95.4
192.168.95.5
... <CORTE> ...
192.168.95.252
192.168.95.253
192.168.95.254
```

Também é possível iterar sobre uma lista de elementos.

```
>>> portList = [21, 22, 25, 80, 110]
>>> for port in portList:
...     print port
...
21
22
25
80
110
```

Com loops aninhados, pode-se obter uma união prática dos exemplos acima.

```
>>> for x in range(1,255):
...     for port in portList:
...             print "[+] 192.168.95."\
...             +str(x)+":"+str(port)
...
    ... <CORTE> ...
[+] 192.168.95.249:21
[+] 192.168.95.249:22
[+] 192.168.95.249:25
[+] 192.168.95.249:80
[+] 192.168.95.249:110
[+] 192.168.95.250:21
[+] 192.168.95.250:22
[+] 192.168.95.250:25
[+] 192.168.95.250:80
[+] 192.168.95.250:110
    ... <CORTE> ...
```

Referências

PYGAME. *About*. 2013. Disponível em: <http://www.pygame.org/wiki/about>. Acesso em: 26.10.2013. Citado na página 20.

PYTHON SOFTWARE FOUNDATION. *Lexical Analisys*. 2013. Disponível em: <http://docs.python.org/2/reference/lexical_analysis.html>. Acesso em: 11.09.2013. Citado na página 8.

PYTHON SOFTWARE FOUNDATION. *Memory Management*: Overview. 2013. Disponível em: <http://docs.python.org/2/c-api/memory.html>. Acesso em: 15.09.2013. Citado na página 12.

PYTHON SOFTWARE FOUNDATION. *re — Regular expression operations*: The re module. 2013. Disponível em: <http://docs.python.org/2/library/re.html>. Acesso em: 21.10.2013. Citado na página 18.

PYTHON SOFTWARE FOUNDATION. *String — Common string operations*: The string module. 2013. Disponível em: <http://docs.python.org/2/library/string.html>. Acesso em: 21.10.2013. Citado na página 18.

ROSSUM, G. van. *Guido van Rossum on the History of Python*. 2011. Vídeo (110 min). Disponível em: <http://www.youtube.com/watch?v=ugqu10JV7dk>. Acesso em: 03.09.2013. Citado na página 6.