Interpretador/Compilador Python

Eduardo Bastos
Universidade Católica de Pelotas
eduardob@ucpel.tche.br

Juliano Freitas
Universidade Católica de Pelotas
jubafre@ucpel.tche.br

Resumo

O presente artigo apresenta uma visão geral da arquitetura do Python, ilustrando o processo das etapas de análise léxica, sintática e geração de código. É realizada uma abordagem sobre a estrutura da arquitetura, contemplando os aspectos da linguagem, interpretador e gramática, e os estilos evidenciados na linguagem, abordando a gerência de memória, interface c/python, reflexão, seqüência de execução de módulos (grupo-seqüêncial) e recursividade.

1. Introdução

A linguagem Python foi desenvolvida pelo holandês Guido Van Rossun, em 1990, com o objetivo inicial de suprir a falta de uma linguagem de script simples e eficiente para o sistema operacional distribuído Amoeba, desenvolvido pela equipe de Andrew Tanenbaum. A idéia de Van Rossum era criar uma evolução da linguagem ABC, de cuja criação havia participado, que fosse atraente para programadores em C/Unix.

Python é uma linguagem de programação interpretada, interativa, orientada a objetos. Incorpora módulos, exceções, tipagem dinâmica, tipos de dados dinâmicos de alto nível, e classes. Combina o poder notável com a sintaxe muito limpa. Tem interfaces a muitas chamadas e bibliotecas de sistema, assim como aos vários sistemas de janela, e é extensivel a C ou C++. É também usada como uma linguagem de extensão para aplicações que necessitam de interfaces programáveis. É portátil: funciona em muitos tipos de UNIX, no mac, e em PCs sob o MSDOS, Windows, Windows NT, e o OS/2

Foi concebida como uma linguagem totalmente orientada a objetos, mas sem impedir que o usuário opte por programar ignorando este fato. Isso quer dizer, que internamente, Python trata tudo como objetos, sejam strings, funções, listas ou números inteiros, mas permite que sejam escritos programas sem a obrigatoriedade do uso de orientação a objetos.

Embora Python seja considerado uma linguagem interpretada como o Perl, Tcl, e algumas outras linguagens, ela emprega um estágio de compilação que traduz scripts Python em uma série de bytecodes, os quais são executados pela Máquina Virtual do Python. O uso dos estágios da compilação e bytecode ajudam a melhorar o desempenho e faz o Python muito mais rápido que interpretadores puros como o BASIC, embora mais lento de linguagens verdadeiramente compiladas, tais como C e Pascal. Entretanto, ao contrário de muitas outras linguagens, as versões do bytecode dos módulos podem ser salvos e executados sem ter que recompilá-los a cada vez que são requeridos, melhorando desse modo a performance, pela eliminação do estágio de compilação. O bytecode criado é completamente independente de sistema operacional e plataforma, bem como o bytecode produzido pelo Java.

2. Estruturas da Arquitetura

Primeiramente é importante observar a distinção entre a linguagem e o interpretador Python. A linguagem é o sistema utilizado ou executado por um programador, já o interpretador é o sistema implementado por um desenvolvedor Python. O termo Python é utilizado neste artigo para referir-se ao interpretador, a menos que seja indicado de outra maneira.

2.1 Interpretação da Arquitetura

A arquitetura do Python pode ser classificada em quatro componentes principais. O diagrama abaixo representa a interpretação da arquitetura do nível mais elevado do sistema Python.

No lado esquerdo, são agrupados módulos do núcleo Python, de bibliotecas e módulos definidos pelos usuários. A biblioteca e os módulos definidos pelos usuários podem estender o sistema Python desde que seja mantida a portabilidade.

No centro do diagrama está ilustrado o núcleo do interpretador Python. As setas dentro da caixa do interpretador indicam fluxo de dados. As setas bidirecionais entre ”Analisador Sintático - Tipo de Objeto”e ”Compilador - Alocador

de Memória"indicam o relacionamento "utiliza". A seta unidirecional entre "Avaliador de Código - Estado atual do Python"indica o relacionamento "modifica". As variações da espessura das linhas servem para melhorar a legibilidade, e não tem nenhum significado conceitual.

Ao lado direito é mostrado o estado atual do Python, alocador de memória e a estrutura de objetos e tipos, que constituem o ambiente em tempo de execução. O estado atual do Python refere-se ao estado de execução do interpretador, visto como uma máquina de estado finito muito grande, complexa, e modificável. O alocador de memória é responsável por alocar a memória para objetos de Python (externo e interno) e é conectado com as rotinas padrão malloc do C. As estruturas de objetos e tipos representam os objetos internos que estão disponíveis em Python.

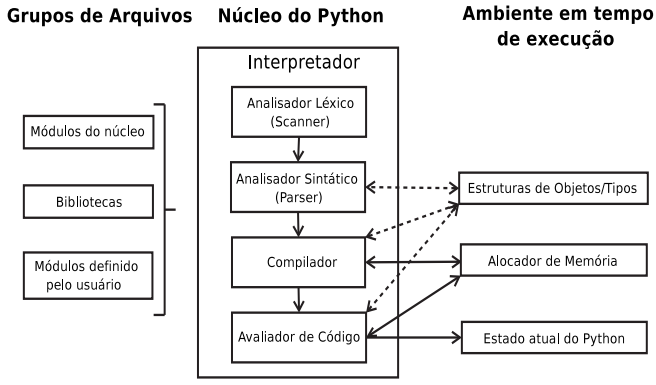

Figura 1. Interpretação da Arquitetura Python

2.2 Disposição de um arquivo

O diagrama abaixo apresenta quatro diretórios principais, os quais descrevem a disposição de um arquivo físico Python. É importante observar que Python usa o termo módulo para significar funcionalidades estendidas, mas, pela convenção do Python, há sempre somente um arquivo em um módulo.

O diretório "Modules"contém todos os módulos escritos em C, incluindo ambos os módulos do núcleo e aqueles que requerem simplesmente uma maior velocidade. Muitos módulos do núcleo (tais como o módulo "OS") são escritos na linguagem Python. O diretório denominado "Parser"é um tanto incorreto, pois contém não somente rotinas do analisador sintático, como também as rotinas do analisador léxico e rotinas para gerar o analisador sintático baseado na gramática da linguagem Python (muito similar ao YACC ou Bison, mas não como Robust). O diretório "Objects"contém todos as rotinas e arquivos de cabeçalhos para implementar objetos em tempo de execução e alguns objetos internos necessários para o interpretador Python funcionar corretamente. Contém todos os tipos internos de objetos Python (built-in objects). Por fim, o diretório "Library"agrupa variadas rotinas padronizadas de bibliotecas.

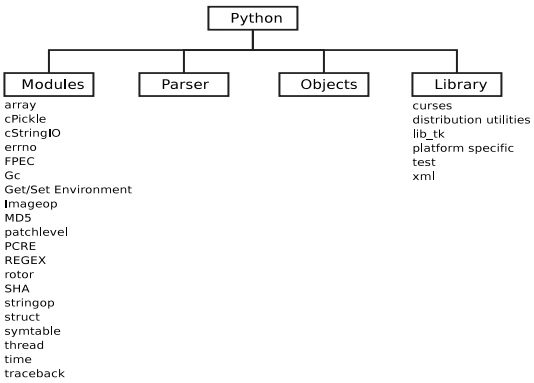

Figura 2. Disposição de um arquivo fisico

2.3 Objetos e Tipos

A linguagem de programação Python possui um modelo universal de objetos, o qual indica que todo e qualquer dado contido em um programa escrito nesta linguagem será parte de um objeto. Como a implementação do núcleo Python foi feita em ANSI C, na verdade as instâncias de objetos são internamente ponteiros para estruturas de dados. A hierarquia principal do gerenciamento de acesso e do desenvolvimento da estrutura de objetos Python é chamada de *PyObject*. Na prática, todos os objetos ativos que serão manipulados durante a execução de um programa possuem um ponteiro interno registrado em *PyObject*.

Esta estrutura possui dois componentes: o próprio ponteiro (que também representa o tipo do objeto) e um contador de referências. O armazenamento do tipo de um objeto também relaciona parâmetros de alocação e métodos de controle que são utilizados principalmente durante a análise da execução dos programas. Esta técnica permite que os colaboradores de Python passem um objeto como um ponteiro a um *PyObject*, alcancem o ponteiro à estrutura do tipo, e usem esta informação para acessar corretamente a estrutura do objeto que contém os valores do objeto, sem sobrecarga de gerenciamento.

O diagrama abaixo ilustra a criação de uma instância do tipo Inteiro. Um *PyObject* é alocado na memória visando a obtenção de um ponteiro para uma estrutura de dados mais complexa, no caso um *IntObject*.

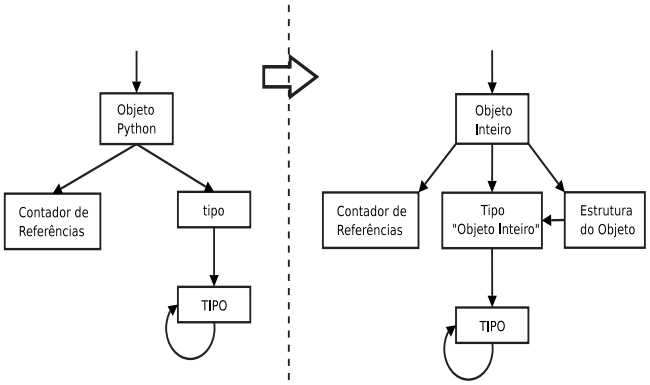

Figura 3. Criação de uma instância do tipo Inteiro

Conceitualmente, todos os objetos em tempo de execução Python pertencem a uma das quatro categorias:

Math respresentações e operações em números

Internal usado principalmente pelo interpretador em tempo de execução, mas pode ser chamado de um programa Python

Composition representação de estruturas conceituais e físicas do programa

Containers agrupamentos lógicos de sub-elementos

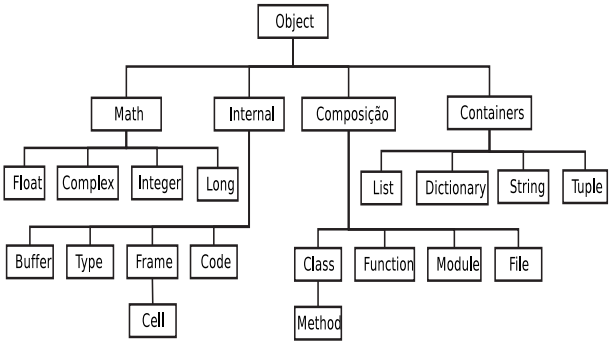

Figura 4. Categorias de Objetos Python

2.4 Interpretador

Em uma visão abstrata, o analisador léxico (lexer/scanner) faz a varredura de um arquivo Python ou de uma única string da linha de comando, e gera uma estrutura com o símbolo (token) que representa o arquivo, passando-a ao analisador sintático. Por sua vez, o analisador sintático recebe este símbolo e gera uma árvore de sintaxe abstrata, denominada AST. O código fonte do Python usa uma terminologia um pouco diferente de um compilador padrão, visto que um compilador padrão adicionaria um nó a AST, e o interpretador Python adiciona um símbolo a AST. O resultado final é o mesmo, já que a estrutura simbólica aponta realmente a uma estrutura do nó.

O analisador sintático chama o compilador com a AST terminada. O compilador examina a AST e gera o código intermediário codificado (bytecode), de maneira similar ao interpretador Java. O compilador chama, então, o avaliador de código que avalia os opcodes do bytecode. Isto, naturalmente, muda o estado do interpretador Python e faz o programa Python executar.

Abaixo algumas diferenças entre o Python e outros compiladores e interpretadores padrões:

- Python faz a varredura do arquivo inteiro, gerando todos os símbolos (tokens) de uma única vez. A maioria dos compiladores e interpretadores têm o analisador sintático que invocam ao analisador léxico, pedindo o símbolo seguinte quando necessitado. Isto tem o efeito lateral que nenhum erro semântico pode ser encontrado a menos que não haja nenhum erro léxico (de sintaxe).
- A manipulação de erro em Python é global e não local. Esta não é necessariamente uma diferença, mas coloca Python firmemente em uma categoria particular da manipulação de erro. A manipulação de erro é uma das partes mais difíceis de construir um bom compilador ou interpretador.
- Python, como um interpretador clássico, não realiza otimização em múltiplas instruções (opcode). Os desenvolvedores Python dizem que o avaliador de código é altamente otimizado, pois mesmo otimizando múltiplas intruções, não resultaria em um ganho de performance. Em um interpretador clássico, a otimização é quase impossível porque somente uma linha do código fonte é interpretada a cada momento.

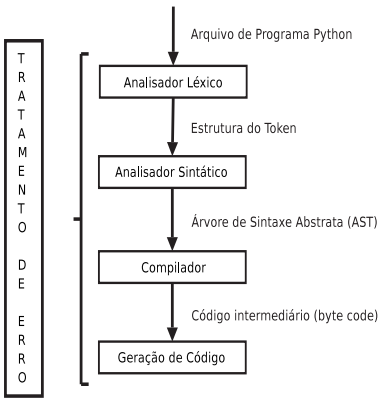

Figura 5. Estruturas do Interpretador

3. Estrutura Léxica

Python usa uma notação modificada da gramática BNF, como descrita em [9]. Cada regra começa por um nome e dois pontos. Uma barra vertical é usada para separar alternativas. O símbolo vezes (*) significa zero ou mais repetições do item precedente; Do mesmo modo, o símbolo mais (+) significa uma ou mais repetições, e uma frase entre colchetes ([]) significa zero ou uma ocorrência, sendo opcional seu uso. Strings literais são incluídas entre aspas. Espaços em branco separam os tokens. Regras são normalmente escritas em uma única linha, porém quando contém alternativas, são formatadas em várias linhas.

3.1 Linhas e Indentação

Um programa python é divido por uma seqüência de linhas lógicas, cada uma composta por uma ou mais linhas físicas, construídas explicitamente ou implicitamente por regras de união de linhas. O final de cada linha lógica é representada pelo token *NEWLINE*; por sua vez, cada final de linha física possui uma convenção dependente da plataforma (em sistemas Unix é utilizado o caracter LF (linefeed); no windows é uma seqüência de CR LF (retorno seguido por um linefeed), já em Macintosh é o caracter CR (retorno)).

Python usa a indentação para expressar estruturas de blocos. Ao contrário de outras linguagens, não faz uso de delimitadores, como chaves e begin/end, mas somente espaços em brancos. São utilizados os tokens *INDENT* e *DEDENT*, e uma pilha. No início de cada linha lógica, o nível da linha é comparado ao topo da pilha. Se for igual, nada acontece. Se for maior, é colocado na pilha, e um token *INDENT* é gerado. Se for menor, retira todos os elementos da pilha até que este se torne o topo da pilha, gerando para cada elemento retirado, um token *DEDENT*. No final do arquivo, um token *DEDENT* é gerado para cada número da pilha maior que zero.

3.2 Token

Cada linha lógica é quebrada em uma série de elementos léxicos, denominados tokens. Cada token pode ser classificado em:

Identificador: Um nome utilizado para identificar variáveis, funções, classes, módulos, ou outros objetos. Começa por uma letra ou por um sobrescrito, seguido por zero ou mais letras, sobrescritos e dígitos. Maíusculas e minúsculas são diferentes, e a pontuação de caracteres não são permitidas. Algumas convenções são usadas: *Classes* são iniciadas por uma letra maíuscula seguida por letras minúsculas; *Identificador Privado* inicia por um simples sobrescrito; *Identificador Protegido* inicia por dois sobrescritos; *Identificador Especial* inicia e termina por dois sobrescritos.

Palavra reservada :

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

Tabela 1. Palavras reservadas da linguagem Python

Operador:

+	-	*	/	%	**	//	<<	>>	&
|	^	~	<	<=	>	>=	<>	!=	=

Tabela 2. Operadores da linguagem Python

Delimitador:

(	)	{	}	[	]
,	:	.	‘	=	;
+=	-=	*=	/=	//=	%=
=	| =	^=	>>=	<<=	**=
’	”	#	\		

Tabela 3. Delimitadores da linguagem Python

Literal:

42	Inteiro
3.14	Ponto-flutuante
1.0J	Imaginário
’str’, ”str”, ”str”	String
[1,2,3]	Lista
(1,2,3)	Tupla
’a’:1, ’b’:2, ’c’:3	Dicionário

Tabela 4. Literais da linguagem Python

4. Estilos da Arquitetura

4.1 Camadas

4.1.1 Gerenciador de Memória

O gerenciador de memória Python utiliza um alocador de objetos, o qual é chamado para cada alocação e desalocação, a menos que os alocadores específicos de objetos implementem um esquema proprietário de alocação (por exemplo, ints), e um coletor de lixo (gargabe collector), o qual opera seletivamente em objetos compostos.

Cada objeto no Python tem uma contagem de referência, sendo incrementada, quando um objeto é alocado pela primeira vez ou uma cópia membro-a-membro é executada, e decrementada, quando as cópias são destruídas ou um objeto é desalocado. Por fim, quando a contagem alcança zero, o coletor de lixo remove o objeto.

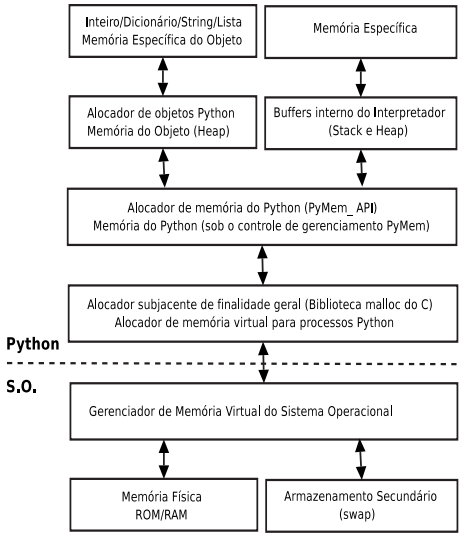

Figura 6. Estrutura do Gerenciador de Memória do Python

4.1.2 C/Python Interface

Um programa em C pode empregar três técnicas para acessar objetos Python.

1. A primeira técnica utiliza a camada denominada Very-High Level Layer (VHLL), fornece o nível mais elevado de controle e é considerada a mais fácil. Um programa C pode usar a VHLL, chamando uma função que passe como parâmetro uma string do código fonte do Python. A string é passada ao interpretador, que usa os objetos built-ins para executá-la. O resultado é retornado para a chamada do programa em C. Essencialmente a VHLL fornece uma interface para acesso do interpretador Python por programas em C.
2. A segunda técnica é o uso de objetos built-ins. Cada um destes objetos define seu próprio tipo de depedência de interface.
3. A terceira técnica é o uso de objetos abstratos, os quais fornecem um padrão para métodos que podem ser aplicados a qualquer objeto concreto. Se um objeto concreto não suportar um método particular, então a camada do objeto abstrato retorna um erro a chamada do programa em C.

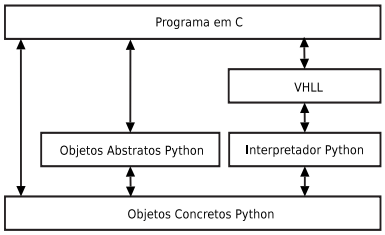

Figura 7. Estrutura da Interface C/Python

4.2 Reflexão

Python não implementa um padrão arquitetural reflexivo puro como especificado por POSA (Pattern-Oriented Software Architecture). A reflexão é evidenciada em um programa que possa mudar sua estrutura ou comportamento em tempo de execução. Será descrito primeiramente a versão POSA de reflexão, e em seguida, apresentado um pequeno exemplo da reflexão na prática da linguagem Python. Finalmente, será descrito a execução que permite potencialidades reflexivas do Python.

4.2.1 Sistema de Reflexão Puro

Em geral, um programa interage com os meta-objetos através do protocolo MOP (Meta-object Protocol). Os objetos básicos (Base-objects) interagem com os meta-objetos e também com outros objetos básicos através dos meta-objetos, e mantém uma informação atual baseada na estrutura e/ou descrições comportamentais mantidas pelos meta-objetos. Cada instância, poderia ter múltiplos níveis. O nível básico poderia interagir diretamente com o meta-objeto (como apresentado) ou através do protocolo MOP. No estilo de camadas, cada camada é independente da camada imediamente inferior. No estilo de reflexão, as camadas são mutuamente dependentes (por exemplo, a camada do meta-objeto depende da camada de objetos básicos, o qual pode depender de outros meta-objetos).

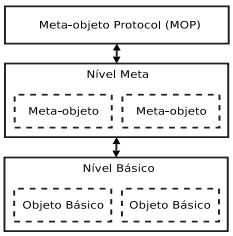

Figura 8. Sistema de Reflexão Puro

4.2.2 Reflexão no Python

Considera-se a implementação Python um subconjunto da reflexão pura. Em Python, o protocolo MOP, meta-objetos e os objetos básicos são combinados em uma única entidade, denominada Python Language Class Object. Nesta implementação, todas as funções para adicionar, modificar e deletar atributos e métodos são encapsuladas no arquivo classobject.c. Em python, é verificado a mudança do comportamento da estrutura do interpretador que corresponde à reflexão, pois a implementação permite a mudança de instâncias de classes de objetos, em tempo de execução.

O exemplo abaixo é capturado de uma sessão inativa do IDLE (ambiente integrado de desenvolvimento em Python), e mostra como podemos mudar a classe de um objeto durante o tempo de execução. O código começa definindo duas classes, ClassA e ClassB, cada uma com métodos diferentes. Um nova instância da ClassA é criada, e então seu tipo é mudado para ClassB, e após de volta para a ClassA.

```
>>> class ClassA:                    # Define ClassA
      def setdata(self, value):
        self.data = value
      def display(self):
        print 'Class A data: %s' % self.data

>>> class ClassB:                    # Define ClassB
      def output(self):
        print 'Class B data: %s ' % self.data

>>> newobject = ClassA()             # Cria uma nova instancia da ClassA
>>> newobject.setdata(23)            # Seta dados para esta instancia com valor 23
>>> newobject.display()              # Usa o método da ClassA
Class A data: 23

>>> newobject.__class__=ClassB       # Muda para objeto da ClassB
>>> newobject.output()               # Usa o método da ClassB
Class B data: 23

>>> newobject.display()              # Método da ClassA não existe
Traceback (most recent call last):   # Retorna erro
  File "", line 1, in ?
    newobject.display()
AttributeError: ClassB instance has no attribute 'display'

>>> newobject.__class__=ClassA       # Objeto volta a ser da ClassA
>>> newobject.display()              # Usa o método da ClassA
Class A data: 23
```

Figura 9. Exemplo de Reflexão no Python

Como pode ser visto, o objeto de uma classe é acessível pelos programas em Python como um atributo class e pode ser mudado simplesmente atribuindo um novo valor ao atributo. Assim na linguagem Python, um programador pode realmente atribuir novamente uma instância do objeto a uma *classe diferente* em tempo de execução.

4.2.3 Como isto funciona

Internamente durante o tempo de execução, cada instância de objeto é representada por uma estrutura denominada *PyInstanceObject*, que contém um ponteiro a uma estrutura chamada *PyClassObject*, a qual representa uma classe e fornece ligações para todos os métodos e atributos associados com a classe e sua hierarquia. Quando um programador define uma classe, o interpretador interage com cada um dos atributos da classe (ou membros de variáveis), adicionando um novo atributo ao *Dicionário de Atributos e Métodos de Classes*. Um objeto instanciado contém um ponteiro para uma cópia do dicionário de classes, e há somente uma cópia de todo o método particular dentro do sistema (NOTA: não há nenhuma variável estática em Python). Assim quando é atribuído um novo valor a instância do atributo da classe, o ponteiro da classe aponta simplesmente a um objeto diferente. Este mecanismo permite a própria modificação do programa para mudar a estrutura e o comportamento em tempo de execução que faz uso da vantagem da reflexão. Eficientemente, o Python (desde a versão 1.5) admite a reflexão na linguagem permitindo o acesso virtualmente transparente às estruturas em tempo de execução subjacentes.

O programador pode criar classes definidas pelo usuário, as quais podem ser completamente originais ou extendidas de alguma classe já existente na linguagem Python. Internamente, uma classe Python segue a seguinte forma de acordo com o diagrama abaixo, onde cada caixa é um *PyObject*. O significado real do *PyObject* está no texto da caixa. As setas significam "aponta para". Para uma instância da *PyClassObject* (que é uma estrutura em C) contém um ponteiro para os três *PyObjects* (dicionário, classes básicas e variáveis internas). Esta é uma versão simplificada da execução real.

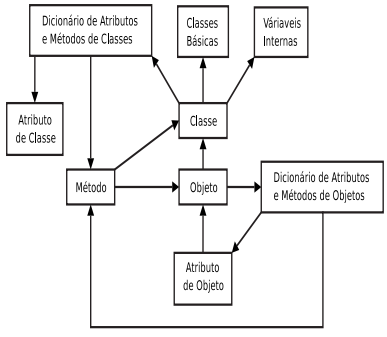

Figura 10. Esquema de Reflexão no Python

4.3 Grupo-Seqüencial

O interpretador principal do Python usa uma Arquitetura de Fluxo de Dados (Data Flow Architecture), onde cada cada componente - analisador léxico, sintático, compilador e avaliador -, pode ser um programa autônomo. Estes componentes são integrados para conseguir um melhor desempenho, mas continuam funcionando de forma seqüencial (Batch Sequential). Cada componente termina sua tarefa, chamando o componente seguinte com as respectivas modificações realizadas. Por exemplo, o analisador sintático faz uma completa análise gramatical na estrutura, transformando-a em uma AST, invocando então o compilador com a AST terminada.

4.4 Programa principal e sub-rotinas

Neste estilo um programa principal é dividido em várias subrotinas menores, as quais são subdivididas em outras subrotinas, até alcançar uma subrotina que consiga resolver facilmente o problema. Na verdade, este estilo é provavelmente o mais familiar de todos, por ser a essência da programação estruturada.

5. Conclusão

Após este estudo verificou-se que a implementação existente do interpretador Python difere de outros, no processo da análise léxica e sintática. Notou-se também uma diversidade de estilos utilizados na programação, como apresentado no tópico 4.

Pôde-se constatar que o Python utiliza uma Máquina Virtual para executar códigos intermediários (bytecodes), assim como outras linguagens interpretadas como Perl e Java.

Alguns aspectos da arquitetura Python e projetos existentes não foram abordados neste artigo. Cita-se o projeto PyPy, que visa construir uma máquina virtual Python utilizando a própria linguagem de programação Python, isto se torna possível devido a reflexão da linguagem.

Por fim, encorajamos membros da comunidade a escreverem mais artigos e tutoriais sobre a arquitetura Python, pois atualmente não encontra-se uma larga documentação destes tópicos, apesar da linguagem ser open source.

6. Referências

[1] Lutz, Mark; Ascher David. Learning Python. Editora O'Reilly, 1999. 382p.

[2] Martelli, Alex; PYTHON IN A NUTSHELL - A desktop Quick and Reference. Editora O'Reilly, 2003. 654p.

[3] ILS O Intérprete e a Escrita. Disponível por WWW em http://www.interpretels.hpg.ig.com.br/7.htm, 2002. (29/10/02).

[4] Indicações históricas. Disponível por www em http://gmc.ucpel.tche.br/python/python-zope/historia.htm, 2002. (5/11/03).

[5] Características diversas da Linguagem Python. Disponível por www em http://gmc.ucpel.tche.br/python/python-zope/caracteristicas-diversas/, 2002. (5/11/03).

[6] Página Oficial da Linguagem. Disponível por www em http://www.python.org, [s.d.]. (5/11/03).

[7] Aspectos Formais da Linguagem Python. Disponível por www em http://lula.dmat.furg.br/ python/aspectos.html, [s.d.].(5/11/03).

[8] The Architecture of Python. Disponível por www em http://wiki.cs.uiuc.edu/cs427/PYTHON, [s.d.]. (5/11/03).

[9] BNF da Linguagem Python. Disponível por www em http://www.python.org/doc/current/ref/grammar.txt, [s.d.]. (5/11/03).