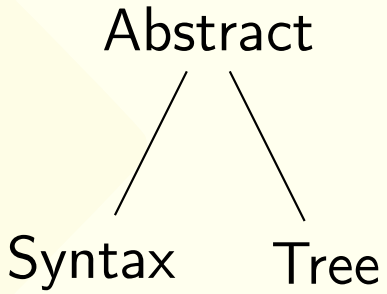
AST

O que é e como usar o módulo padrão?

Danilo J. S. Bellini
@danilobellini

PYTHON SUDESTE

2024-07-07



Python Sudeste 2024 @ UFSCar/SP

O que acontece com o código Python até a execução?

Podemos dividir a conversão do código fonte em 4 passos:

1. Código fonte (uma *string*)
2. Sequência de *tokens* resultantes do analisador léxico (`INDENT`/`DEDENT`, identificadores, literais, operadores, `NEWLINE`, [soft] keywords, e delimitadores)
3. **Árvore sintática** resultante do parser
4. *Bytecode* Python compilado (insumo do interpretador)

Por que chamamos de “abstrata”?

Uma árvore sintática é dita “abstrata” quando ela “descarta” alguma informação mesmo quando esta faz parte da gramática da linguagem, mas que não é relevante de ser armazenada por:

- Ser implícita na estrutura da própria árvore; ou
- Não possuir valor semântico a ser interpretado, isto é, não ter relevância na tradução realizada no processo de compilação.

Por exemplo, parênteses opcionais, comentários, linhas em branco e “`;`” separando *statements* não trazem nenhum valor para o *bytecode* resultante. O “`:`” que denota o início de um novo bloco está sempre implícito como parte da sintaxe dos respectivos blocos, não é preciso que esse detalhe de sintaxe tenha um “nó” próprio para representá-lo.

Um *Hello world* com AST!

```
>>> import ast
>>> código = "print('Python Sudeste em São Carlos!')"

# ast.parse -> Cria o objeto AST a partir do código
>>> root = ast.parse(código)  # mode="exec" implícito
>>> type(root)
<class 'ast.Module'>

# ast.dump -> Permite "enxergar" a AST
>>> print(ast.dump(root, indent=2))
Module(
  body=[
    Expr(
      value=Call(
        func=Name(id='print', ctx=Load()),
        args=[
          Constant(value='Python Sudeste em São Carlos!')],
        keywords=[]))],
  type_ignores=[])

# compile -> Pode ser usado para converter AST em bytecode
>>> code_obj = compile(root, filename="<string>", mode="exec")
>>> type(code_obj)  # Este tipo é o types.CodeType
<class 'code'>

# exec/eval -> Permitem rodar o bytecode
>>> exec(code_obj)
Python Sudeste em São Carlos!
```

"Raiz" da AST gerada: os 4 modos

A gramática abstrata do Python define para o elemento raiz `mod`:

```
mod = Module(stmt* body, type_ignore* type_ignores)
    | Interactive(stmt* body)
    | Expression(expr body)
    | FunctionType(expr* argtypes, expr returns)
```

`mode`	Tipo devolvido por `ast.parse`	Conteúdo do `body`
`"exec"`	`ast.Module`	Lista de nós (*statements*)
`"single"`	`ast.Interactive`	Lista de nós (uma linha)
`"eval"`	`ast.Expression`	Um único nó (expressão)
`"func_type"`	`ast.FunctionType`	Não há (legado)

Podemos ver o resultado de `ast.dump` com `python -m ast`:

```
$ python -m ast -m eval <<< '2 + 2'
Expression(
   body=BinOp(
      left=Constant(value=2),
      op=Add(),
      right=Constant(value=2)))
```

Expressões VS *statements*; `ast.Expression` VS `ast.Expr`

Os nós da AST são objetos que herdam de `ast.AST`.

Uma expressão é algo que em Python produz um valor resultante, mesmo que este seja **`None`**. (`expr` na gramática).

Um *statement* (`stmt` na gramática) é uma operação "atômica" no código, tal como uma definição de função ou uma atribuição. Todo código Python consiste em *statements*.

O nó de tipo `ast.Expression` serve apenas como raiz do modo `eval`, as expressões são na realidade os valores que podem constar em seu campo `body`.

Uma expressão pode ser usada como um *statement*, por exemplo em uma chamada de função. O `ast.Expr` serve para esse fim, contendo em seu campo `value` a expressão propriamente dita, convertendo-a em `statement`, para a finalidade sintática.

O modo `single` e o legado `func_type`

O modo `single` unifica o "EP" do REPL (*read-eval-print-loop*), fazendo chamadas ao `sys.displayhook` para os nós `ast.Expr` que constarem no `body`. Isto é, os resultados das "expressões como *statements*" são exibidos como se tudo fosse digitado no terminal do Python em uma única linha lógica (um ou mais *statements* separados por "`;`").

O modo `func_type` é um legado (Python < 3.5) contendo anotações de tipo para funções presentes em comentários como `# type: (int, complex) -> str`.

```
$ python -m ast -m "func_type" <<< '(int, complex) -> str'
FunctionType(
   argtypes=[
      Name(id='int', ctx=Load()),
      Name(id='complex', ctx=Load())],
   returns=Name(id='str', ctx=Load()))
```

O *built-in* `compile`

Embora exibido anteriormente para compilar AST para *bytecode*, esse built-in tradicionalmente recebe como entrada código Python como uma *string*. Em particular, seu quarto argumento posicional, `flags` pode ser atribuído a `ast.PyCF_ONLY_AST`, nesse caso o comportamento do `compile` será o mesmo do `ast.parse` (converter código fonte Python em AST).

Quando usado para compilar AST para *bytecode*, o modo fornecido para o `compile` deve ser o mesmo usado para criar a AST, do contrário teremos um `TypeError`.

Casos de uso para AST

- Avaliação de características de um código (análise estática)
  e.g. com `ast.NodeVisitor` ou `ast.walk`
  - Garantia de segurança/isolamento (`ast.literal_eval`)
  - Teste de arquitetura
  - Garantia de equivalência entre códigos (*formatter*)
  - Cálculo de complexidade ciclomática (McCabe)
- Transformação de um código (e.g. via `ast.NodeTransformer`)
  - Remoção de *statements* de um bloco (e.g. **`assert`**)
  - Coleta de um fragmento de um arquivo, (e.g. para o `setup.py` obter o `__version__` sem importar o módulo/pacote)
  - Substituição de um fragmento de uma expressão
  - Inserção de *statements* artificiais
  - Macro sintática
- Geração de código em tempo de execução (AST sintética)
  - Metaprogramação
  - Otimização

Exemplo de síntese: DynagRPC v0.1!

Biblioteca criada para tornar a implementação de servidores gRPC "menos não-pythonico". Código criado para um cliente, aberto com autorização para a realização da presente palestra.

A classe `GrpcTypeCastRegistry` implementa um registro de conversores entre objetos *protobuf* e objetos de tipos nativos do Python (principalmente dicionários) utilizando síntese de AST.

`https://github.com/danilobellini/dynagrpc`

Na síntese, a "posição no código" (`lineno` e `col_offset`) precisa ser preenchida nos nós da AST, e há funções como `ast.copy_location` e `ast.increment_lineno` feitas para esse fim. Esse problema foi resolvido da maneira mais simples, com o `ast.fix_missing_locations`, que atribui recursivamente os nós com o mesmo valor do *parent* (antecessor imediato, nó "pai").

FIM!

https://github.com/danilobellini/slides-latex

Referências:

- https://docs.python.org/3/library/ast.html
- https://github.com/danilobellini/dynagrpc
- https://greentreesnakes.readthedocs.io