Python. Лекция 14. Устройство интерпретатора языка Python.

Лексический анализ

Лексический анализатор языка программирования разбивает исходный текст программы (состоящий из одиночных символов) на лексемы - неделимые "слова" языка.

Основные категории лексем Python: идентификаторы и ключевые слова (NAME), литералы (STRING, NUMBER и т.п.), операции (OP), разделители, специальные лексемы для обозначения (изменения) отступов (INDENT, DEDENT) и концов строк (NEWLINE), а также комментарии (COMMENT). Лексический анализатор доступен через модуль `tokenize`, а определения кодов лексем содержатся в модуле `token` стандартной библиотеки Python. Следующий пример показывает лексический анализатор в действии:

```
import StringIO, token, tokenize

        prog_example = """
        for i in range(100): # comment
          if i % 1 == 0: \
          print ":", t**2
        """.strip()

        rl = StringIO.StringIO(prog_example).readline

        for t_type, t_str, (br,bc), (er,ec), logl in tokenize.generate_tokens(rl):
          print "%3i %10s : %20r" % (t_type, token.tok_name[t_type], t_str)
```

А вот что выведет эта программа, разбив на лексемы исходный код примера:

```
prog_example:

          1       NAME :                'for'
          1       NAME :                  'i'
          1       NAME :                 'in'
          1       NAME :              'range'
         50         OP :                  '('
          2     NUMBER :                '100'
         50         OP :                  ')'
         50         OP :                  ':'
         52    COMMENT :          '# comment'
          4    NEWLINE :                 '\n'
          5     INDENT :                  '  '
          1       NAME :                 'if'
          1       NAME :                  'i'
         50         OP :                  '%'
          2     NUMBER :                  '1'
         50         OP :                 '=='
          2     NUMBER :                  '0'
         50         OP :                  ':'
          1       NAME :              'print'
          3     STRING :                '":"'
         50         OP :                  ','
          1       NAME :                  't'
         50         OP :                 '**'
          2     NUMBER :                  '2'
          6     DEDENT :                   ''
          0  ENDMARKER :                   ''
```

Фактически получен поток лексем, который может использоваться для различных целей. Например, для синтаксического "окрашивания" кода на языке Python. Словарь `token.tok_name` позволяет получить мнемонические имена для типа лексемы по номеру.

Синтаксический анализ

Вторая стадия преобразования исходного текста программы в байт-код интерпретатора состоит в синтаксическом анализе исходного текста. Модуль `parser` содержит функции `suite()` и `expr()` для построения **деревьев синтаксического разбора** соответственно для кода программ и выражений Python. Модуль `symbol` содержит номера символов грамматики Python, словарь для получения названия символа из грамматики Python.

Следующая программа анализирует достаточно простой код Python (`prg`) и порождает дерево синтаксического разбора (AST-объект), который тут же можно превращать в кортеж и красиво выводить функцией `pprint.pprint()`. Далее определяется функция для превращения номеров символов в их мнемонические обозначения (имена) в грамматике:

```
import pprint, token, parser, symbol

            prg = """print 2*2"""

            pprint.pprint(parser.suite(prg).totuple())

            def pprint_ast(ast, level=0):
              if type(ast) == type(()):
                for a in ast:
                  pprint_ast(a, level+1)
              elif type(ast) == type(""):
                print repr(ast)
              else:
                print " "*level,
                try:
                  print symbol.sym_name[ast]
                except:
                  print "token."+token.tok_name[ast],

            print
            pprint_ast(parser.suite(prg).totuple())
```

Эта программа выведет следующее (структура дерева отражена отступами):

```
(257,
             (264,
              (265,
               (266,
                (269,
                 (1, 'print'),
                 (292,
                  (293,
                   (294,
                    (295,
                     (297,
                      (298,
                       (299,
                        (300,
                         (301,
                          (302,
                           (303, (304, (305, (2, '2')))),
                           (16, '*'),
                           (303, (304, (305, (2, '2'))))))))))))))),
               (4, ''))),
              (0, ''))

              file_input
               stmt
                simple_stmt
                 small_stmt
                  print_stmt
                   token.NAME 'print'
                   test
                    and_test
                     not_test
                      comparison
                       expr
                        xor_expr
                         and_expr
                          shift_expr
                           arith_expr
                            term
                             factor
                              power
                               atom
                                token.NUMBER '2'
                             token.STAR '*'
                             factor
                              power
                               atom
                                token.NUMBER '2'
                 token.NEWLINE ''
                token.ENDMARKER ''
```

Получение байт-кода

После того как получено дерево синтаксического разбора, компилятор должен превратить его в байт-код, подходящий для исполнения интерпретатором. В следующей программе проводятся отдельно синтаксический анализ, компиляция и выполнение (вычисление) кода (и выражения) в языке Python:

```
import parser

            prg = """print 2*2"""
            ast = parser.suite(prg)
            code = ast.compile('filename.py')
            exec code

            prg = """2*2"""
            ast = parser.expr(prg)
            code = ast.compile('filename1.py')
            print eval(code)
```

Функция `parser.suite()` (или `parser.expr()`) возвращает AST-объект (дерево синтаксического анализа), которое методом `compile()` компилируется в Python байт-код и сохраняется в кодовом объекте `code`. Теперь этот код можно выполнить (или, в случае выражения - вычислить) с помощью оператора `exec` (или функции `eval()`).

Здесь необходимо заметить, что недавно в Python появился пакет `compiler`, который объединяет модули для работы анализа исходного кода на Python и генерации кода. В данной лекции он не рассматривается, но те, кто хочет глубже изучить эти процессы, может обратиться к документации по Python.

Изучение байт-кода

Для изучения байт-кода Python-программы можно использовать модуль `dis` (сокращение от "дизассемблер"), который содержит функции, позволяющие увидеть байт-код в мнемоническом виде. Следующий пример иллюстрирует эту возможность:

```
>>> def f():
            ...   print 2*2
            ...
            >>> dis.dis(f)
              2           0 LOAD_CONST               1 (2)
                          3 LOAD_CONST               1 (2)
                          6 BINARY_MULTIPLY
                          7 PRINT_ITEM
                          8 PRINT_NEWLINE
                          9 LOAD_CONST               0 (None)
                         12 RETURN_VALUE
```

Определяется функция `f()`, которая должна вычислить и напечатать значение выражения `2*2`. Функция `dis()` модуля `dis` выводит код функции `f()` в виде некого "ассемблера", в котором байт-код Python представлен мнемоническими именами. Следует заметить, что при интерпретации используется стек, поэтому `LOAD_CONST` кладет значение на вершину стека, а `BINARY_MULTIPLY` берет со стека два значения и помещает на стек результат их перемножения. Функция без оператора `return` возвращает значение `None`. Как и в случае с кодами для микропроцессора, некоторые байт-коды принимают параметры.

Мнемонические имена можно увидеть в списке `dis.opname` (ниже печатаются только задействованные имена):

```
>>> import dis
            >>> [n for n in dis.opname if n[0] != "<"]
            ['STOP_CODE', 'POP_TOP', 'ROT_TWO', 'ROT_THREE', 'DUP_TOP', 'ROT_FOUR',
            'NOP', 'UNARY_POSITIVE', 'UNARY_NEGATIVE', 'UNARY_NOT', 'UNARY_CONVERT',
            'UNARY_INVERT', 'LIST_APPEND', 'BINARY_POWER', 'BINARY_MULTIPLY',
            'BINARY_DIVIDE', 'BINARY_MODULO', 'BINARY_ADD', 'BINARY_SUBTRACT',
            'BINARY_SUBSCR', 'BINARY_FLOOR_DIVIDE', 'BINARY_TRUE_DIVIDE',
            'INPLACE_FLOOR_DIVIDE', 'INPLACE_TRUE_DIVIDE', 'SLICE+0', 'SLICE+1',
            'SLICE+2', 'SLICE+3', 'STORE_SLICE+0', 'STORE_SLICE+1', 'STORE_SLICE+2',
            'STORE_SLICE+3', 'DELETE_SLICE+0', 'DELETE_SLICE+1', 'DELETE_SLICE+2',
            'DELETE_SLICE+3', 'INPLACE_ADD', 'INPLACE_SUBTRACT', 'INPLACE_MULTIPLY',
            'INPLACE_DIVIDE', 'INPLACE_MODULO', 'STORE_SUBSCR', 'DELETE_SUBSCR',
            'BINARY_LSHIFT', 'BINARY_RSHIFT', 'BINARY_AND', 'BINARY_XOR', 'BINARY_OR',
            'INPLACE_POWER', 'GET_ITER', 'PRINT_EXPR', 'PRINT_ITEM', 'PRINT_NEWLINE',
            'PRINT_ITEM_TO', 'PRINT_NEWLINE_TO', 'INPLACE_LSHIFT', 'INPLACE_RSHIFT',
            'INPLACE_AND', 'INPLACE_XOR', 'INPLACE_OR', 'BREAK_LOOP', 'LOAD_LOCALS',
            'RETURN_VALUE', 'IMPORT_STAR', 'EXEC_STMT', 'YIELD_VALUE', 'POP_BLOCK',
            'END_FINALLY', 'BUILD_CLASS', 'STORE_NAME', 'DELETE_NAME',
            'UNPACK_SEQUENCE', 'FOR_ITER', 'STORE_ATTR', 'DELETE_ATTR', 'STORE_GLOBAL',
            'DELETE_GLOBAL', 'DUP_TOPX', 'LOAD_CONST', 'LOAD_NAME', 'BUILD_TUPLE',
            'BUILD_LIST', 'BUILD_MAP', 'LOAD_ATTR', 'COMPARE_OP', 'IMPORT_NAME',
            'IMPORT_FROM', 'JUMP_FORWARD', 'JUMP_IF_FALSE', 'JUMP_IF_TRUE',
            'JUMP_ABSOLUTE', 'LOAD_GLOBAL', 'CONTINUE_LOOP', 'SETUP_LOOP',
            'SETUP_EXCEPT', 'SETUP_FINALLY', 'LOAD_FAST', 'STORE_FAST', 'DELETE_FAST',
            'RAISE_VARARGS', 'CALL_FUNCTION', 'MAKE_FUNCTION', 'BUILD_SLICE',
            'MAKE_CLOSURE', 'LOAD_CLOSURE', 'LOAD_DEREF', 'STORE_DEREF',
            'CALL_FUNCTION_VAR', 'CALL_FUNCTION_KW', 'CALL_FUNCTION_VAR_KW',
            'EXTENDED_ARG']
```

Легко догадаться, что LOAD означает загрузку значения в стек, STORE - выгрузку, PRINT - печать, BINARY - бинарную операцию и т.п.

Отладка

В интерпретаторе языка Python заложены возможности отладки программ, а в стандартной поставке имеется простейший **отладчик** - `pdb`. Следующий пример показывает программу, которая подвергается отладке, и типичную сессию отладки:

```
# File myfun.py
            def fun(s):
              lst = []
              for i in s:
                lst.append(ord(i))
              return lst
```

Так может выглядеть типичный процесс отладки:

```
>>> import pdb, myfun
            >>> pdb.runcall(myfun.fun, "ABCDE")
            > /examples/myfun.py(4)fun()
            -> lst = []
            (Pdb) n
            > /examples/myfun.py(5)fun()
            -> for i in s:
            (Pdb) n
            > /examples/myfun.py(6)fun()
            -> lst.append(ord(i))
            (Pdb) l
              1     #!/usr/bin/python
              2     # File myfun.py
              3     def fun(s):
              4       lst = []
              5       for i in s:
              6  ->     lst.append(ord(i))
```

```
  7       return lst
[EOF]
(Pdb) p lst
[]
(Pdb) p vars()
{'i': 'A', 's': 'ABCDE', 'lst': []}
(Pdb) n
> /examples/myfun.py(5)fun()
-> for i in s:
(Pdb) p vars()
{'i': 'A', 's': 'ABCDE', 'lst': [65]}
(Pdb) n
> /examples/myfun.py(6)fun()
-> lst.append(ord(i))
(Pdb) n
> /examples/myfun.py(5)fun()
-> for i in s:
(Pdb) p vars()
{'i': 'B', 's': 'ABCDE', 'lst': [65, 66]}
(Pdb) r
- Return -
> /examples/myfun.py(7)fun()->[65, 66, 67, 68, 69]
-> return lst
(Pdb) n
[65, 66, 67, 68, 69]
>>>
```

Интерактивный отладчик вызывается функцией `pdb.runcall()` и на его приглашение `(Pdb)` следует вводить команды. В данном примере сессии отладки были использованы некоторые из следующих команд: `l` (печать фрагмента трассируемого кода), `n` (выполнить все до следующей строки), `s` (сделать следующий шаг, возможно, углубившись в вызов метода или функции), `p` (печать значения), `r` (выполнить все до возврата из текущей функции).

Разумеется, некоторые интерактивные оболочки разработчика для Python предоставляют функции отладчика. Кроме того, отладку достаточно легко организовать, поставив в ключевых местах программы, операторы `print` для вывода интересующих параметров. Обычно этого достаточно, чтобы локализовать проблему. В CGI-сценариях можно использовать модуль `cgitb`, о котором говорилось в одной из предыдущих лекций.

Профайлер

Для определения мест в программе, на выполнение которых уходит значительная часть времени, обычно применяется **профайлер**.

Модуль profile

Этот модуль позволяет проанализировать работу функции и выдать статистику использования процессорного времени на выполнение той или иной части алгоритма.

В качестве примера можно рассмотреть профилирование функции для поиска строк из списка, наиболее похожих на данную. Для того чтобы качественно профилировать функцию `difflib.get_close_matches()`, нужен большой объем данных. В файле russian.txt собрано 160 тысяч слов русского языка. Следующая программа поможет профилировать функцию `difflib.get_close_matches()`:

```
import difflib, profile

                def print_close_matches(word):
                  print      "\n".join(difflib.get_close_matches(word      +      "\n",
open("russian.txt")))

                profile.run(r'print_close_matches("профайлер")')
```

При запуске этой программы будет выдано примерно следующее:

```
провайдер

 трайлер

 бройлер

 899769 function calls (877642 primitive calls) in 23.620 CPU seconds

 Ordered by: standard name

 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
 1    0.000    0.000   23.610   23.610 <string>:1(?)
 1    0.000    0.000   23.610   23.610 T.py:6(print_close_matches)
 1    0.000    0.000    0.000    0.000 difflib.py:147(__init__)
 1    0.000    0.000    0.000    0.000 difflib.py:210(set_seqs)
 159443    1.420    0.000    1.420    0.000 difflib.py:222(set_seq1)
 2    0.000    0.000    0.000    0.000 difflib.py:248(set_seq2)
 2    0.000    0.000    0.000    0.000 difflib.py:293(__chain_b)
 324261    2.240    0.000    2.240    0.000 difflib.py:32(_calculate_ratio)
 28317    1.590    0.000    1.590    0.000 difflib.py:344(find_longest_match)
 6474    0.100    0.000    2.690    0.000 difflib.py:454(get_matching_blocks)
 28317/6190    1.000    0.000    2.590    0.000 difflib.py:480(__helper)
 6474    0.450    0.000    3.480    0.001 difflib.py:595(ratio)
 28686    0.240    0.000    0.240    0.000 difflib.py:617(<lambda>)
 158345    8.690    0.000    9.760    0.000 difflib.py:621(quick_ratio)
 159442    2.950    0.000    4.020    0.000 difflib.py:650(real_quick_ratio)
 1    4.930    4.930   23.610   23.610 difflib.py:662(get_close_matches)
 1    0.010    0.010   23.620   23.620 profile:0(print_close_matches("профайлер"))
 0    0.000             0.000          profile:0(profiler)
```

Здесь колонки таблицы показывают следующие значения: ncalls - количество вызовов (функции), tottime - время выполнения кода функции (не включая времени выполнения вызываемых из нее функций), percall - то же время, в пересчете на один вызов, cumtime - суммарное время выполнения функции (и всех вызываемых из нее функций), filename - имя файла, lineno - номер строки в файле, function - имя функции (если эти параметры известны).

Из приведенной статистики следует, что наибольшие усилия по оптимизации кода необходимо приложить в функциях `quick_ratio()` (на нее потрачено 8,69 секунд), `get_close_matches()` (4,93 секунд), затем можно заняться `real_quick_ratio()` (2,95 секунд) и `_calculate_ratio()` (секунд).

Это лишь самый простой вариант использования профайлера: модуль `profile` (и связанный с ним `pstats`) позволяет получать и обрабатывать статистику: их применение описано в документации.

Модуль timeit

Предположим, что проводится оптимизация небольшого участка кода. Необходимо определить, какой из вариантов кода является наиболее быстрым. Это можно сделать с помощью модуля `timeit`.

В следующей программе используется метод `timeit()` для измерения времени, необходимого для вычисления небольшого фрагмента кода. Измерения проводятся для

трех вариантов кода, делающих одно и то же: конкатенирующих десять тысяч строк в одну строку. В первом случае используется наиболее естественный, "лобовой" прием инкрементной конкатенации, во втором - накопление строк в списке с последующим объединением в одну строку, в третьем применяется списковое включение, а затем объединение элементов списка в одну строку:

```
from timeit import Timer

                t = Timer("""
                res = ""
                for k in range(1000000,1010000):
                  res += str(k)
                """)
                print t.timeit(200)

                t = Timer("""
                res = []
                for k in range(1000000,1010000):
                  res.append(str(k))
                res = ",".join(res)
                """)
                print t.timeit(200)

                t = Timer("""
                res = ",".join([str(k) for k in range(1000000,1010000)])
                """)
                print t.timeit(200)
```

Разные версии Python дадут различные результаты прогонов:

```
# Python 2.3
                77.6665899754
                10.1372740269
                9.07727599144

                # Python 2.4
                9.26631307602
                9.8416929245
                7.36629199982
```

В старых версиях Python рекомендуемым способом конкатенации большого количества строк являлось накопление их в списке с последующим применением функции `join()` (кстати, инкрементная конкатенация почти в восемь раз медленнее этого приема). Начиная с версии 2.4, инкрементная конкатенация была оптимизирована и теперь имеет даже лучший результат, чем версия со списками (которая вдобавок требует больше памяти). Но чемпионом все-таки является работа со списковым включением, поэтому свертывание циклов в списковое включение позволяет повысить эффективность кода.

Если требуются более точные результаты, рекомендуется использовать метод `repeat(n, k)` - он позволяет вызывать `timeit(k)` `n` раз, возвращая список из `n` значений. Необходимо отметить, что на результаты может влиять загруженность компьютера, на котором проводятся испытания.

Оптимизация

Основная реализация языка Python пока что не имеет оптимизирующего компилятора, поэтому разговор об оптимизации касается только **оптимизации кода** самим программистом. В любом языке программирования имеются свои характерные приемы оптимизации кода. Оптимизация (улучшение) кода может происходить в двух (зачастую конкурирующих) направлениях: скорость и занимаемая память. В условиях достатка оперативной памяти приложения обычно оптимизируют по скорости. При оптимизации по времени программы для одноразового вычисления следует иметь в

виду, что в общее время решения задачи входит не только выполнение программы, но и время ее написания. Не стоит тратить усилия на оптимизацию программы, если она будет использоваться очень редко.

Следует учитывать, что программа, реализующая некоторый алгоритм, не может быть оптимизирована до бесконечно малого времени вычисления: используемый алгоритм имеет определенную **временную сложность** и программу, основанную на слишком сложном алгоритме, существенно оптимизировать не удастся. Можно попытаться сменить алгоритм (хотя многие задачи этого сделать не позволяют) или ослабить требования к решениям. Иногда помогает упрощение алгоритма. К сожалению, оптимизация кода, как и программирование - задача неформальная, поэтому умение оптимизировать код приходит с опытом.

Если скорость работы программы при большой длине данных не устраивает, следует поискать более эффективный алгоритм. Если же более эффективный алгоритм практически нецелесообразен, можно попытаться провести оптимизацию кода.

Собственно, в данном примере для модуля `timeit` уже показан практический способ нахождения оптимального кода. Стоит также отметить, что с помощью профайлера нужно определить места кода, отнимающие наибольшую часть времени. Обычно это действия, выполняемые в самом вложенном цикле. Можно попытаться вынести из цикла все, что можно вычислить в более внешнем цикле или вообще вне цикла.

В языке Python вызов функции является относительно дорогостоящей операцией, поэтому на критичных по скорости участках кода следует избегать вызова большого числа функций.

В некоторых случаях работу программы на Python можно ускорить в несколько раз с помощью специального оптимизатора (он не входит в стандартную поставку Python, но свободно распространяется): `psyco`. Для ускорения программы достаточно добавить следующие строки в начале главного модуля программы:

```
import psyco
                psyco.full()
```

Правда, некоторые функции не поддаются "компиляции" с помощью `psyco`. В этих случаях будут выданы предупреждения. Посмотрите документацию по `psyco` с тем, чтобы узнать ограничения в его использовании и способы их преодоления.

Еще одним вариантом ускорения работы приложения является переписывание критических участков алгоритма на языках более низкого уровня (C/C++) и использование модулей расширения из Python. Однако эта крайняя мера обычно не требуется или модули для задач, требующих большей эффективности, уже написаны. Например, для работы с растровыми изображениями имеется прекрасная библиотека модулей PIL (Python Imaging Library). Численные расчеты можно выполнять с помощью пакета Numeric и т.д.

Pychecker

Одним из наиболее интересных инструментов для анализа исходного кода Python программы является Pychecker. Как и lint для языка C, Pychecker позволяет выявлять слабости в исходном коде на языке Python. Можно рассмотреть следующий пример с использованием Pychecker:

```
import re, string
            import re
            a = "a b c"
```

```
def test(x, y):
  from string import split
  a = "x y z"
  print split(a) + x

test(['d'], 'e')
```

Pychecker выдаст следующие предупреждения:

```
badcode.py:1: Imported module (string) not used
              badcode.py:2: Imported module (re) not used
              badcode.py:2: Module (re) re-imported
              badcode.py:5: Parameter (y) not used
              badcode.py:6: Using import and from ... import for (string)
              badcode.py:7: Local variable (a) shadows global defined on line 3
              badcode.py:8: Local variable (a) shadows global defined on line 3
```

В первой строке импортирован модуль, который далее не применяется, то же самое с модулем `re`. Кроме того, модуль `re` импортирован повторно. Другие проблемы с кодом: параметр y не использован; модуль `string` применен как в операторе `import`, так и во `from-import` ; локальная переменная `a` затеняет глобальную, которая определена в третьей строке.

Можно переписать этот пример так, чтобы Pychecker выдавал меньше предупреждений:

```
import string
              a = "a b c"

              def test(x, y):
                a1 = "x y z"
                print string.split(a1) + x

              test(['d'], 'e')
```

Теперь имеется лишь одно предупреждение:

```
goodcode.py:4: Parameter (y) not used
```

Такое тоже бывает. Программист должен лишь убедиться, что он не сделал ошибки.

Исследование объекта

Даже самые примитивные объекты в языке программирования Python имеют возможности, общие для всех объектов: можно получить их уникальный идентификатор (с помощью функции `id()`), представление в виде строки - даже в двух вариантах (функции `str()` и `repr()`); можно узнать атрибуты объекта с помощью встроенной функции `dir()` и во многих случаях пользоваться атрибутом `__dict__` для доступа к словарю имен объекта. Также можно узнать, сколько других объектов ссылается на данный с помощью функции `sys.getrefcount()`. Есть еще сборка мусора, которая применяется для освобождения памяти от объектов, которые более не используются, но имеют ссылки друг на друга (циклические ссылки). Сборкой мусора (garbage collection) можно управлять из модуля `gc`.

Все это подчеркивает тот факт, что объекты в Python существуют не сами по себе, а являются частью системы: они и их отношения строго учитываются интерпретатором.

Сразу же следует оговориться, что Python имеет две стороны интроспекции: "официальную", которую поддерживает описание языка и многие его реализации, и "неофициальную", которая использует особенности той или иной реализации. С

помощью "официальных" средств интроспекции можно получить информацию о принадлежности объекта тому или иному классу (функция `type()`), проверить принадлежность экземпляра классу (`isinstance()`), отношение наследования между классами (`issubclass()`), а также получить информацию, о которой говорилось чуть выше. Это как бы приборная доска машины. С помощью "неофициальной" интроспекции (это то, что под капотом) можно получить доступ к чему угодно: к текущему фрейму исполнения и стеку, к байт-коду функции, к некоторым механизмам интерпретатора (от загрузки модулей до полного контроля над внутренней средой исполнения). Сразу же стоит сказать, что этот механизм следует рассматривать (и тем более вносить изменения) очень деликатно: разработчики языка не гарантируют постоянство этих механизмов от версии к версии, а некоторые полезные модули используют эти механизмы для своих целей. Например, упомянутый ранее ускоритель выполнения Python-кода `psyco` очень серьезно вмешивается во фреймы исполнения, заменяя их своими объектами. Кроме того, разные реализации Python могут иметь совсем другие внутренние механизмы.

Сказанное стоит подкрепить примерами.

В первом примере исследуется объект с помощью "официальных" средств. В качестве объекта выбрана обычная строка:

```
>>> s = "abcd"
            >>> dir(s)
            ['__add__', '__class__', '__contains__', '__delattr__', '__doc__',
            '__eq__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
            '__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__',
            '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
            '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
            '__str__', 'capitalize', 'center', 'count', 'decode',
            'encode', 'endswith', 'expandtabs', 'find', 'index', 'isalnum', 'isalpha',
            'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
            'lower', 'lstrip', 'replace', 'rfind', 'rindex', 'rjust', 'rstrip',
            'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
            'translate', 'upper', 'zfill']
            >>> id(s)
            1075718400
            >>> print str(s)
            abcd
            >>> print repr(s)
            'abcd'
            >>> type(s)
            <type 'str'>
            >>> isinstance(s, basestring)
            True
            >>> isinstance(s, int)
            False
            >>> issubclass(str, basestring)
            True
```

"Неофициальные" средства интроспекции в основном работают в области представления объектов в среде интерпретатора. Ниже будет рассмотрено, как главная (на настоящий момент) реализация Python может дать информацию об определенной пользователем функции:

```
>>> def f(x, y=0):
            ...   """Function f(x, y)"""
            ...   global s
            ...   return t + x + y
            ...
            >>> f.secure = 1     # присваивается  дополнительный атрибут
            >>> f.func_name      # имя
            'f'
            >>> f.func_doc       # строка документации
            'Function f(x, y)'
            >>> f.func_defaults  # значения по умолчанию
```

```
(0,)
>>> f.func_dict      # словарь атрибутов функции
{'secure': 1}
>>> co = f.func_code # кодовый объект
>>> co
<code object f at 0x401ec7e0, file "<stdin>", line 1>
```

Кодовые объекты имеют свои атрибуты:

```
>>> co.co_code       # байт-код
't\x00\x00|\x00\x00\x17|\x01\x00\x17Sd\x01\x00S'
>>> co.co_argcount   # число аргументов
2
>>> co.co_varnames   # имена переменных
('x', 'y')
>>> co.co_consts     # константы
(None,)
>>> co.co_names      # локальные имена
('t', 'x', 'y')
>>> co.co_name       # имя блока кода (в нашем случае - имя функции)
'f'
```

и так далее. Более правильно использовать для получения всех этих сведений модуль `inspect`.

Модуль inspect

Основное назначение модуля `inspect` - давать приложению информацию о модулях, классах, функциях, трассировочных объектах, фреймах исполнения и кодовых объектах. Именно модуль `inspect` позволяет заглянуть "на кухню" интерпретатора Python.

Модуль имеет функции для проверки принадлежности объектов различным типам, с которыми он работает:

Функция	**Проверяемый тип**
inspect.isbuiltin	Встроенная функция
inspect.isclass	Класс
inspect.iscode	Код
inspect.isdatadescriptor	Описатель данных
inspect.isframe	Фрейм
inspect.isfunction	Функция
inspect.ismethod	Метод
inspect.ismethoddescriptor	Описатель метода
inspect.ismodule	Модуль
inspect.isroutine	Функция или метод
inspect.istraceback	Трассировочный объект

Пример:

```
>>> import inspect
                >>> inspect.isbuiltin(len)
                True
                >>> inspect.isroutine(lambda x: x+1)
                True
                >>> inspect.ismethod(''.split)
                False
                >>> inspect.isroutine(''.split)
                True
                >>> inspect.isbuiltin(''.split)
                True
```

Объект типа модуль появляется в Python-программе благодаря операции импорта. Для получения информации о модуле имеются некоторые функции, а объект-модуль обладает определенными атрибутами, как продемонстрировано ниже:

```
>>> import inspect
               >>> inspect.ismodule(inspect)
               True
               >>> inspect.getmoduleinfo('/usr/local/lib/python2.3/inspect.pyc')
               ('inspect', '.pyc', 'rb', 2)
               >>> inspect.getmodulename('/usr/local/lib/python2.3/inspect.pyc')
               'inspect'
               >>> inspect.__name__
               'inspect'
               >>> inspect.__dict__
               . . .
               >>> inspect.__doc__
               "Get useful information from live Python objects.\n\nThis module encapsulates
               .. . .
```

Интересны некоторые функции, которые предоставляют информацию об исходном коде объектов:

```
>>> import inspect
               >>> inspect.getsourcefile(inspect)   # имя файла исходного кода
               '/usr/local/lib/python2.3/inspect.py'
               >>> inspect.getabsfile(inspect)      # абсолютный путь к файлу
               '/usr/local/lib/python2.3/inspect.py'
               >>> print inspect.getfile(inspect)   # файл кода модуля
               /usr/local/lib/python2.3/inspect.pyc
               >>> print inspect.getsource(inspect)  # исходный текст модуля (в виде строки)
               # -*- coding: iso-8859-1 -*-
               """Get useful information from live Python objects.
               . . .
               >>> import smtplib
               >>> # Комментарий непосредственно перед определением объекта:
               >>> inspect.getcomments(smtplib.SMTPException)
               '# Exception classes used by this module.\n'
               >>> # Теперь берем строку документирования:
               >>> inspect.getdoc(smtplib.SMTPException)
               'Base class for all exceptions raised by this module.'
```

С помощью модуля `inspect` можно узнать состав аргументов некоторой функции с помощью функции `inspect.getargspec()`:

```
>>> import inspect
                >>> def f(x, y=1, z=2):
                ...   return x + y + z
                ...
                >>> def g(x, *v, **z):
                ...   return x
                ...
                >>> print inspect.getargspec(f)
                (['x', 'y', 'z'], None, None, (1, 2))
                >>> print inspect.getargspec(g)
                (['x'], 'v', 'z', None)
```

Возвращаемый кортеж содержит список аргументов (кроме специальных), затем следуют имена аргументов для списка позиционных аргументов (*) и списка именованных аргументов (**), после чего - список значений по умолчанию для последних позиционных аргументов. Первый аргумент-список может содержать вложенные списки, отражая структуру аргументов:

```
>>> def f((x1,y1), (x2,y2)):
...   return 1
...
>>> print inspect.getargspec(f)
([['x1', 'y1'], ['x2', 'y2']], None, None, None)
```

Классы (как вы помните) - тоже объекты, и о них можно кое-что узнать:

```
>>> import smtplib
>>> s = smtplib.SMTP
>>> s.__module__  # модуль, в котором был определен объект
'smtplib'
>>> inspect.getmodule(s)  # можно догадаться о происхождении объекта
<module 'smtplib' from '/usr/local/lib/python2.3/smtplib.pyc'>
```

Для визуализации дерева классов может быть полезна функция `inspect.getclasstree()`. Она возвращает иерархически выстроенный в соответствии с наследованием список вложенных списков классов, указанных в списке-параметре. В следующем примере на основе списка всех встроенных классов-исключений создается дерево их зависимостей по наследованию:

```
import inspect, exceptions

def formattree(tree, level=0):
  """Вывод дерева наследований.
  tree - дерево, подготовленное с помощью inspect.getclasstree(),
  которое представлено списком вложенных списков и кортежей.
  В кортеже entry первый элемент - класс, а второй - кортеж с его
  базовыми классами. Иначе entry - вложенный список.
  level - уровень отступов
  """
  for entry in tree:
    if type(entry) is type(()):
      c, bases = entry
      print level * "  ", c.__name__, \
            "(" + ", ".join([b.__name__ for b in bases]) + ")"
    elif type(entry) is type([]):
      formattree(entry, level+1)

v = exceptions.__dict__.values()
exc_list = [e for e in v
            if inspect.isclass(e) and issubclass(e, Exception)]

formattree(inspect.getclasstree(exc_list))
```

С помощью функции `inspect.currentframe()` можно получить текущий фрейм исполнения. Атрибуты фрейма исполнения дают информацию о блоке кода, исполняющегося в точке вызова метода. При вызове функции (и в некоторых других ситуациях) на стек кладется соответствующий этому фрейму блок кода. При возврате из функции текущим становится фрейм, хранившийся в стеке. Фрейм содержит контекст выполнения кода: пространства имен и некоторые другие данные. Получить эти данные можно через атрибуты фреймового объекта:

```
import inspect

def f():
  fr = inspect.currentframe()
  for a in dir(fr):
    if a[:2] != "__":
      print a, ":", str(getattr(fr, a))[:70]
```

```
f()
```

В результате получается

```
f_back : <frame object at 0x812383c>
               f_builtins : {'help': Type help() for interactive help, or help(object) for help ab
               f_code : <code object f at 0x401d83a0, file "<stdin>", line 11>
               f_exc_traceback : None
               f_exc_type : None
               f_exc_value : None
               f_globals : {'f': <function f at 0x401e0454>, '__builtins__': <module '__builtin__
               f_lasti : 68
               f_lineno : 16
               f_locals : {'a': 'f_locals', 'fr': <frame object at 0x813c34c>}
               f_restricted : 0
               f_trace : None
```

Здесь `f_back` - предыдущий фрейм исполнения (вызвавший данный фрейм), `f_builtins` - пространство встроенных имен, как его видно из данного фрейма, `f_globals` - пространство глобальных имен, `f_locals` - пространство локальных имен, `f_code` - кодовый объект (в данном случае - байт-код функции `f()`), `f_lasti` - индекс последней выполнявшейся инструкции байт-кода, `f_trace` - функция трассировки для данного фрейма (или `None`), `f_lineno` - текущая строка исходного кода, `f_restricted` - признак выполнения в ограничительном режиме.

Получить информацию о стеке интерпретатора можно с помощью функции `inspect.stack()`. Она возвращает список кортежей, в которых есть следующие элементы:

```
(фрейм-объект, имя_файла, строка_в_файле, имя_функции,
                список_строк_исходного_кода, номер_строки_в_коде)
```

Трассировочные объекты также играют важную роль в интроспективных возможностях языка Python: с их помощью можно отследить место возбуждения исключения и обработать его требуемым образом. Для работы с трассировками предусмотрен даже специальный модуль - `traceback`.

Трассировочный объект представляет содержимое стека исполнения от места возбуждения исключения до места его обработки. В обработчике исключений связанный с исключением трассировочный объект доступен посредством функции `sys.exc_info()` (это третий элемент возвращаемого данной функцией кортежа).

Трассировочный объект имеет следующие атрибуты:

- `tb_frame` Фрейм исполнения текущего уровня.
- `tb_lineno` и `tb_lasti` Номер строки и инструкции, где было возбуждено исключение.
- `tb_next` Следующий уровень стека (другой трассировочный объект).

Одно из наиболее частых применений модуля `traceback` - "мягкая" обработка исключений с выводом отладочной информации в удобном виде (в лог, на стандартный вывод ошибок и т.п.):

```
#!/usr/bin/python

               def dbg_except():
                 """Функция для отладки операторов try-except"""
                 import traceback, sys, string
                 print sys.exc_info()
                 print " ".join(traceback.format_exception(*sys.exc_info()))

               def bad_func2():
                 raise StandardError
```

```
def bad_func():
  bad_func2()

try:
  bad_func()
except:
  dbg_except()
```

В результате получается примерно следующее:

```
(<class exceptions.StandardError at 0x4019729c>,
                <exceptions.StandardError instance at 0x401df2cc>,
                <traceback object at 0x401dcb1c>)
                Traceback (most recent call last):
                   File "pr143.py", line 17, in ?
                    bad_func()
                   File "pr143.py", line 14, in bad_func
                    bad_func2()
                   File "pr143.py", line 11, in bad_func2
                    raise StandardError
                 StandardError
```

Функция `sys.exc_info()` дает кортеж с информацией о возбужденном исключении (класс исключения, объект исключения и трассировочный объект). Элементы этого кортежа передаются как параметры функции `traceback.format_exception()`, которая и печатает информацию об исключении в уже знакомой форме. Модуль `traceback` содержит и другие функции (о них можно узнать из документации), которые помогают форматировать те или иные части информации об исключении.

Разумеется, это еще не все возможности модуля `inspect` и свойств интроспекции в Python, а лишь наиболее интересные функции и атрибуты. Подробнее можно прочитать в документации или даже в исходном коде модулей стандартной библиотеки Python.

Заключение

С помощью возможностей интроспекции удается рассмотреть фазы работы транслятора Python: лексический анализ, синтаксический разбор и генерации кода для интерпретатора, саму работу интерпретатора можно видеть при помощи отладчика.

Вместе с тем, в этой лекции было дано представление об использовании профайлера для исследования того, на что больше всего тратится процессорное время в программе, а также затронуты некоторые аспекты оптимизации Python-программ и варианты оптимизации кода на Python по скорости.

Наконец, интроспекция позволяет исследовать не только строение программы, но и объектов, с которыми работает эта программа. Были рассмотрены возможности Python по получению информации об объектах - этом основном строительном материале, из которого складываются данные любой Python-программы.