Parythony Design Specs

Data Structures

- Parythony : Node Size = 12 bytes
- Node List Size = 256 nodes/page x 12 B/node = 3072 B/page
- Page List Size = 512 page slots x (4-byte ptr. + 24-bit next addr + updated byte) = 4096 bytes
- Chapter List Size = 2048 page lists x 4 B/page list = 8192 bytes
- Address Space = 32 (4-byte ptr.) + 4 (16 bits/node) = 36 bits = 64 GB
- Page Addr: 4-bit book no + 11-bit chapter no + 9-bit page idx = 24 bits
- Addr Value: 24-bit page addr + 8-bit node idx = 32 bits
- Node Types:
    - Object Ref Node: 0 bit, 31-bit refcount, 0 bit, 31-bit root (attr) ptr., code ptr.
    - Object Value Node: 0 bit, 31-bit refcount, 1 bit, header, 4-byte value
    - Lisp Node: 1 bit, 31-bit refcount, 2 x 4-byte ptrs. to lisp, obj ref, obj val nodes
    - Tree Node: 2 x 16-bit hdrs., 2 x 4-byte values
    - Stack Node: 4-byte hdr., 4-byte value, 4-byte next
    - Base Node: prior base ptr., root (parm/loc) ptr., next ptr.
    - Long Node: 64-bit long, double, or set
    - Callback Node: obj ref, code ptr.
    - ByteZero Node: 12 bytes, null-terminated
    - String Node: 3 x 4-byte Unicode chars.
    - Array Node: same as stack node, used for bytezero, string, set values
    - String List: bytezero value, may contain newline chars.
    - Dict Leaf Node: string list, array node (list of values)
- Header Values:
    - node, boolean, int, long, float, double, object, lisp, bytezero, string, set, list, dict, dict leaf, callback, op, paren, null
- Binary Tree: has root, max path size = 32 bits
    - object node
    - base node
    - list, dict
- Class Inheritance:
    - Code ptr. may point to method in ancestor class
    - Current class includes all ancestor attributes
- Page Types: chapter list, page list, node list (up to 16 chapter lists exist)
- Swap File:
    - 16 chapter lists (2048 ptrs. x 4 B/ptr.) = 128K
    - 16 x 2048 page lists (512 page slots x 8 B/slot) = 128 MB
    - 16 x 2048 x 512 node lists (3072 B + 24-bit page addr + 8-bit node idx) = 48 GB + 64 MB
        - page addr = prevEmpty page addr
        - node idx = first empty node
- Page Swapping:
    1. Occurs when accessing null ptr. (swapped page) in page list
    2. currIdx = idx of swapped page in page list
    3. **if** next(currIdx) = 0 and firstEmpty = currIdx **then** go ahead
    4. **else** go to Page Refresh
    5. oldIdx = firstFull
    6. prevEmpty = 0
    7. **if** updated(oldIdx) **then** write prevEmpty + old node list to swap file

8. new ptr. = addr. of old node list
9. Read prevEmpty + node list (of currIdx) from swap file
10. Overwrite old node list with data read
11. **if** prevEmpty = 0 **then** firstEmpty = next(currIdx)
12. **else** next(prevEmpty) = next(currIdx)
13. pageList[currIdx] = new ptr.
14. next(currIdx) = firstFull
15. firstFull = currIdx
16. updated(oldIdx) = 0
17. updated(currIdx) = 0

- Page Refreshing:
  1. Occurs when accessing null ptr. (swapped page) in page list
  2. currIdx = idx of swapped page in page list
  3. **if** next(currIdx) != 0 or firstEmpty != currIdx **then** go ahead
  4. **else** go to Page Swap
  5. Read prevEmpty + node list (of currIdx) from swap file
  6. **if** prevEmpty = 0 **then** firstEmpty = next(currIdx)
  7. **else** next(prevEmpty) = next(currIdx)
  8. new ptr. = new node list
  9. pageList[currIdx] = new ptr.
  10. **if** lastFull = 0 **then** lastFull = currIdx
  11. **else** next(lastFull) = currIdx
  12. next(currIdx) = 0
  13. updated(currIdx) = 0
- Page Updating: set updated(currIdx) = 1
- Tree-balancing functionality needed
- Little or no support for arrays
- Reference counting used for garbage collection

Memory Usage

- 1 K-node = 4 pages x 256 nodes/page = 1024 nodes
- Memory/user (lo) = 256 pages/user x 3072 B/page = 0.75 MB/user = 768K/user
- Memory/user (hi) = 1024 pages/user x 3072 B/page = 3 MB/user
- Memory/server = 1 GB minimum
- Users/server (lo) = 1024 MB/server x 0.33 users/MB = 341 users/server
- Users/server (hi) = 1024 MB/server x 1.33 users/MB = 1365 users/server
- Disk space/server = 1024 GB
- Disk space/user (lo) = 1024 GB/server x (0.75/1024 GB/user) = 0.75 GB/user/server
- Disk space/user (hi) = 1024 GB/server x (3/1024 GB/user) = 3 GB/user/server
- Nodes/user (lo) = 0.75 MB/user x (1/16 nodes/B) = 48 K-nodes/user = 192 pages
- Nodes/user (hi) = 3 MB/user x (1/16 nodes/B) = 192 K-nodes/user = 768 pages
- Slots/day = 2 slots/hour x 24 hours/day = 48 slots/day
- User-slots/server/day = 341 users/server x 48 slots/day = 16,384
- Users/subscriber (lo) = 100
- Users/subscriber (hi) = 200
- Subscribers/server (lo) = 16,384 user-slots/server/day x (0.01 subscribers/user) = 164
- Subscribers/server (hi) = 16,384 user-slots/server/day x (0.005 subscribers/user) = 82
- Revenue/server/year (lo) = 82 subscribers/server x $20/subscriber/year = $1640/year
- Revenue/server/year (hi) = 164 subscribers/server x $20/subscriber/year = $3280/year

Code Execution

All Parythony source code is in Polish notation, in which operators precede their operands. The following algorithm is used, in which operators are stored in one stack and operands in a separate stack. Executable code consists of tree nodes.

```
rightp = root
while true do
  if rightp = 0 then
    op = pop operator
    if op = root then
      return true
    if op = while/for/loopbody then
      pop rightp from operator stack
      continue
    if op = if then
      pop rightp from operator stack
      pop (
      continue
    if op = block then
      pop (
      pop if from operator stack
      pop (
      pop rightp from operator stack
      continue
    count = 0
    while true do
        pop operand
        if open parenthesis then break
        push operand on operator stack
        increment count
    if op = call then
      rightp = handlecall(count)
      continue
    if op = constructor then
      rightp = handlecons(count)
      continue
    if op = callback then
      rightp = handlecallback(count)
      continue
    pop operand from operator stack
    push operand
    repeat count - 1 times
        pop operand from operator stack
        push operand
        rightpop = pop
        leftpop = pop
        push op(leftpop, rightpop)
        // (: obj attridx) => obj...
    if count = 1 then
      if unary op then
        push op(pop)
      else
```

```
      rightpop = pop
      leftpop = pop
      push op(leftpop, rightpop)
  pop rightp from operator stack
  continue
currnode = getnode(rightp)
if open parenthesis then
  push on operand stack
  push rightp on operator stack
  rightp = currnode.downp
else if operand then
  push on operand stack
  rightp = currnode.rightp
else if operator then
  push on operator stack
  rightp = currnode.rightp
else if funcbody then
  handlebody
  rightp = currnode.rightp
else if endfunc then
  pop downto begin from operator stack
  pop rightp from operator stack
else if while/for then
  rightp = currnode.rightp
  push rightp, while/for on operator stack
else if do then
  flag = pop
  if not flag then
    pop while, rightp from operator stack
    pop rightp from operator stack
    pop (
else if continue then
  pop downto while from operator stack
  pop rightp from operator stack
else if break then
  pop downto while from operator stack
  pop rightp, rightp from operator stack
  pop (
else if breakfor then
  pop downto for from operator stack
  pop rightp, rightp from operator stack
  pop (
  pop (
else if contfor then
  pop downto loopbody from operator stack
  pop rightp from operator stack
else if then then
  flag = pop
  if flag then
    rightp = currnode.rightp
  else
    pop if from operator stack
    pop (
```

```
      pop rightp from operator stack
  else
    return false

pop downto x from operator stack:
  pop multiple from operator stack
  if: pop (
  while: pop (

do block while flag:
  while true do block if not flag then break

handlecons(count):
  pop classref from operator stack
  gen objref: root 0/1 = instance/class vars
  push objref on operator stack
  return handlecall(count)

handlecall(count):
  pop objref from operator stack
  push objref
  pop codept from operator stack
  return handlecodept(codept, count)

handlecodept(codept, count):
  repeat count - 2 times
    pop val from operator stack
    push val
  push count - 1
  return codept

handlecallback(count):
  pop callback from operator stack
  unpack objref, codept
  push objref
  return handlecodept(codept, count)

handlebody:
  count = pop
  root = new node
  for i = count - 2 downto 0 do
    parm = pop
    add parm to 1st half of tree[i]
  objref = parm
  rightp = currnode.rightp
  loccount = currnode value
  repeat loccount times
    add null node to 2nd half of tree
  rightp = currnode.rightp
```

Parythony Language Features

Parythony is both a subset and superset of Python in which all operators precede their operands, except a given class can only inherit from a single base class (no multiple inheritance). Parythony borrows the interface feature from Java, in which the interface keyword is renamed to "scool". Two other new keywords include "scoolref" (built-in function returns a scool reference) and "abclass" (abstract class). File extensions include .PRY (source code), .PRYC (compiled code), and .PRYML (text file which may have Parythony code embedded in it).

Keyboard Aid

This optional feature enables hyphens, open parentheses, and close parentheses to be entered by typing single quotes, commas, and periods, respectively. When enabled, keyboard aid can be temporarily suppressed by using the Ctrl key in conjunction with typing single quotes, commas, and periods (no character substitution takes place). By convention, hyphens are used to separate words in multi-word identifiers, but single quotes are easier to type than hyphens. Similarly, commas and periods are easier to type than parentheses.

Special Characters

- `()` grouping
- `-_` used in identifiers
- `;` end of stmt.
- `:` dot operator
- `" '` string delimiters
- `\` escape char.
- `#` comment
- `{* *}` block comment

Grammar Notation

- Non-terminal symbol: <symbol name>
- Optional text in brackets: [*text*]
- Repeats zero or more times: [*text*]…
- Repeats one or more times: <symbol name>…
- Pipe separates alternatives: *opt1* | *opt2*
- Comments in *italics*
- Advanced features flagged as **
- Omitted features ***

Parythony Grammar

White space occurs between tokens (parentheses and semicolon need no adjacent white space):

<source file>:
- [<line-comment>] [<stmt-semi>]… [<def>]… [<class>]… [<stmt-semi>]…

<import stmt>:
    import <module>
    import (<module>…)
    from <rel module> import <mod list>
    from <rel module> import all

<module>:
    <name>
    (<name> as <name>)
    (**:** <name>… [as <name>])

<mod list>:
    <id as>
    (<id as>…)

<id as>:
    <mod id>
    (<mod id> as <name>)

<mod id>:
    <mod name>
    <class name>
    <func name>
    <var name>

<rel module>:
    (<colon list> [<name>]...)

<class>:
- (<cls typ> <name> [<base class>] [<does>]
  do <def>…)
- (scool <name> [<does>] do [<const decl>]...
  [<def hdr>]...)

<cls typ>:
  class
  abclass

<does>:
  does (<scool name>...)

<scool name>:
<base class>:
  <name>
  (**:** <name><name>…)

<const decl>:
  const <name><const expr> **;**

<def hdr>:
  def <name> ([<parm>]…)

<def>:
  def <name> ([<parm>]…) do <block>

<parm>:
  <name>
  (set <name><expr>)
  (all <name>)

<block>:
  (<stmt-semi>…)

<stmt-semi>:
  <stmt> **;**

<stmt>:
  pass
  ** <if stmt>
  <while stmt>
  ** <for stmt>
  ** <try stmt>
  <disruptive stmt>
  <call stmt>
  <asst stmt>
  <del stmt>
  <print stmt>
  <import stmt>

<disruptive stmt>:
  <continue stmt>
  <break stmt>
  <return stmt>
  ** <raise stmt>

<call expr>:
- (<name> [<expr>]…)
- (**:** <obj expr> [<colon expr>]…
  (<method name> [<expr>]…))
- (call <expr>…)

<call stmt>:
- (<name> [<expr>]…)
- **:** <obj expr> [<colon expr>]…
  (<method name> [<expr>]…)
- call <expr>…

<colon expr>:
  <name>
  (<name> [<expr>]…)

<asst stmt>:
  <asst op><name><expr>
  <asst op><target expr><expr>

<asst op>:
  set | addset | minusset | mpyset | divset |
  idivset | modset | shlset | shrset |
  andset | orset | xorset

<target expr>:
  (**:** <name> [<colon expr>]… <name>)
  (slice <arr><expr> [<expr>])
  (slice <arr><expr> all)

<arr>: *// string, list or tuple*
  <name>
  <expr>

<if stmt>:
- if <expr><block> [elif <expr><block>]…
  [else <block>]

<while stmt>:
  while <expr> do <block>
  do <block> while <expr>

<for stmt>:
  for <name> in <expr> do <block>

<try stmt>:
- try <block> <except clause>… [else
  <block>] [finally <block>]
- try <block> finally <block>

<except clause>:
  except <name> [as <name>] <block>

<raise stmt>:
  raise [<expr> [from <expr>]]

<return stmt>:
    return [<expr>]

<break stmt>:
    break

<continue stmt>:
    continue

<del stmt>:
    del <expr>

<print stmt>: *// built-in func*
    (print [<expr>]…)
    (echo [<expr>]…)

<expr>:
    <keyword const>
    <literal>
    <name>
    (<unary op><expr>)
    (<bin op><expr><expr>)
    (<multi op><expr><expr>…)
    (quest <expr><expr><expr>)
    <list expr>
    *** <tuple expr>
    <bytearray expr>
    <set expr>
    <dict expr>
    <target expr>
    <obj expr>

<obj expr>:
    <name>
    <call stmt>

<unary op>:
    minus *negate*
    notbits *bitwise not*
    not

<bin op>:
    <arith op>
    <comparison op>
    <shift op>
    <bitwise op>
    <boolean op>

<arith op>:
    div | idiv | mod | mpy | add | minus

<comparison op>:
    ge | le | gt | lt | eq | ne | is | in

<shift op>:
    shl | shr

<bitwise op>:
    andbits | orbits | xorbits

<boolean op>:
    and | or | xor

<multi op>:
    mpy | add | or | and |
    strdo *% operator*
    strcat *+ operator*

<const expr>:
    <literal>
    <keyword const>

<literal>:
    <num lit>
    <string lit>
    <bytes lit>

<list expr>:
    (list [<expr>]…)

<tuple expr>:
    (tuple [<expr>]…)

<bytearray expr>:
    (bytearray <expr>)

<set expr>:
    (listuniq <expr>)

<dict expr>:
    (dict [<pair>]…)

<pair>:
    (<name><expr>)
    (<literal><expr>)

<lisp funcs>:
    (quote <expr>)
    (compile <expr>)
    (exec <expr>)

The quote *function takes a block of code and treats it like data, with an implied* list *keyword after every open parenthesis. The* compile *function compiles the output of the* quote *function, which is then executed by the* exec *function.*

No white space allowed between tokens, for rest of Parythony Grammar:

<white space>:
    <white token>...

<white token>:
    <rest-comment>
    <non-line-comment>

<rest-comment>:
    <non-line-comment><line-comment>

<line-comment>:
    # [<char>]... <new-line>

<non-line-comment>:
    <white char>
    <blk-comment>

<blk-comment>:
    {* [<char>]... *}

<white char>:
    <space> | <tab> | <new-line>

<name>:
- [<underscore>]… <letter> [<alnum>]…
  [<hyphen-alnum>]… [<underscore>]…

<hyphen-alnum>:
    <hyphen><alnum>…

<alnum>:
    <letter>
    <digit>

In plain English, names begin and end with zero or more underscores. In between is a letter followed by zero or more alphanumeric characters. Names may also contain hyphens, where each hyphen is preceded and succeeded by an alphanumeric character.

<num lit>:
    <dec int>
    <oct int>
    <hex int>
    <bin int>
    <float>

<dec int>:
    0
    [<hyphen>] <any digit except 0> [<digit>]…

<float>:
    <dec int><fraction> [<exponent>]
    <dec int><exponent>

<fraction>:
    <dot> [<digit>]…

<exponent>:
    <e> [<sign>] <digit>…
<e>:
    e | E

<sign>:
    + | -

<oct int>:
    0o <octal digit>…

<hex int>:
    0x <hex digit>…
    0X <hex digit>…

<bin int>:
    0b <zero or one>…
    0B <zero or one>…

<octal digit>:
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

<hex digit>:
    <digit>
    A | B | C | D | E | F
    a | b | c | d | e | f

<keyword const>:
    None
    True
    False

<colon list>:
    : [**:**]…

<size funcs>:
    (long <expr>)
    (short <expr>)

<string lit>:
    [<str prefix>] <short long>

<str prefix>:
    r | u | R | U

<short long>:
    ' [<short item>]... '
    " [<short item>]... "
    ' ' ' [<long item>]... ' ' '
    " " " [<long item>]... " " "

<short item>:
    <short char>
    <escaped str char>

<long item>:
    <long char>
    <escaped str char>

<short char>:
    any source char. except "\", newline, or
    end quote

<long char>:
    any source char. except "\"

<bytes lit>:
    <byte prefix><shortb longb>

<byte prefix>: *// any case/order*
    b | br

<shortb longb>:
    ' [<shortb item>]... '
    " [<shortb item>]... "
    ' ' ' [<longb item>]... ' ' '
    " " " [<longb item>]... " " "

<shortb item>:
    <shortb char>
    <escaped char>

<longb item>:
    <longb char>
    <escaped char>

<shortb char>:
    any ASCII char. except "\", newline, or
    end quote

<longb char>:
    any ASCII char. except "\"

<escaped char>:
    \newline
        ignore "\", newline chars.
    \\ *backslash*
    \" *double quote*
    \' *single quote*
    \a *bell*
    \b *backspace*
    \f *formfeed*
    \n *new line*
    \r *carriage return*
    \t *tab*
    \v *vertical tab*
    \ooo *octal value = ooo*
    \xhh *hex value = hh*

<escaped str char>:
    <escaped char>
    \N{name} *Unicode char. = name*
    \uxxxx *hex value (16-bit) = xxxx*
    \Uxxxxxxxx
        hex value (32-bit) = xxxxxxxx

Semicolon Grammar

The semicolon switch (the default) allows statements normally enclosed in parentheses to instead be terminated with semicolons, with no need for whitespace surrounding semicolons (similar to parentheses). What follows assumes the switch is off, and statements are enclosed in parentheses.

<source file>:
- [<line-comment>] [<stmt>]… [<def>]… [<class>]…[<stmt>]…

<import stmt>:
(import <module>)
(import (<module>…))
(from <rel module> import <mod list>)
(from <rel module> import all)

<def>:
(def <name> ([<parm>]…) do <stmt>…)

<block>:
(<stmt>…)

<raise stmt>:
(raise [<expr> [from <expr>]])

<return stmt>:
return
(return <expr>)

<if stmt>:
- (if <expr><block> [elif <expr><block>]… [else <block>])

<while stmt>:
(while <expr> do <stmt>…)
(do <block> while <expr>)

<for stmt>:
(for <name> in <expr> do <stmt>…)

<try stmt>:
- (try <block> <except clause>… [else <block>] [finally <block>])
- (try <block> finally <block>)

<call stmt>:
- (<name> [<expr>]…)
- (**:** <obj expr> [<colon expr>]… (<method name> [<expr>]…))
- (call <expr>…)

<asst stmt>:
(<asst op><name><expr>)
(<asst op><target expr><expr>)

<del stmt>:
(del <expr>)

<print stmt>:
(print [<expr>]…)
(echo [<expr>]…)