

PYTHON

By

Brad Touesnard

CS4613 – Programming Languages

Fredericton, New Brunswick

18 December 2003

UNIVERSITY OF NEW BRUNSWICK

FACULTY OF COMPUTER SCIENCE

Table of Contents

Table of Contents	i
1.0 History.....	1
2.0 Language Elements and Rules	1
2.1 Names	1
2.2 Binding.....	2
2.3 Data Types	2
2.4 Typing and Type Checking.....	3
2.5 Scope.....	4
2.6 Variable and Object Lifetime.....	4
3.0 Supported Programming Styles	5
4.0 Distinct Features	5
5.0 Domains	6
6.0 Current Status.....	6
7.0 Conclusion	7
References.....	8
Appendix.....	9

1.0 History

In 1983, Guido van Rossum joined a team of developers working on a language called *ABC* at the National Research Institute for Mathematics and Computer Science (CWI) in the Netherlands. “ABC was intended to be a programming language that could be taught to intelligent computer users who were not computer programmers or software developers in any sense.” (1) In 1986, van Rossum moved on to another project at CWI. The Amoeba project was a distributed operating system. In the late 1980’s, van Rossum and his team found the need for a scripting language for the Amoeba operating system. It was at this point that van Rossum recalled his experience with ABC and began development of a new language that became Python.

Using the parts of ABC that he liked, van Rossum developed a simple virtual machine, parser, and runtime around a basic syntax design. He used indentation for encapsulating statements and chose a small number of flexible data types. Although similar to ABC in many ways, Python presented important differences that van Rossum considered shortfalls with the ABC language. van Rossum considers his “most innovative contribution to Python’s success was making it easy to extend.” (1) That is, making Python extendable allowed developers to add modules for increased functionality. This capability has enabled the expansion of the Python library and extension modules and has allowed the language to adapt to the ever changing programming landscape.

2.0 Language Elements and Rules

2.1 Names

An identifier or name is used to identify a variable, function, class, module, or other object. Names in Python begin with an uppercase letter, lowercase letter, or underscore followed by zero or more letters, underscores, and/or digits. Other characters are not allowed in names. Python is case-sensitive, meaning uppercase and lowercase letters are distinct.

Conventionally, Python developers start class names with an uppercase letter and other identifiers such as variables with lowercase letters. Starting an identifier with an underscore generally implies it is private and starting with two underscores indicates a strongly private identifier. If a name also ends with two underscores, the identifier is a special name reserved by the language.

Python also has twenty eight keywords reserved for special purposes such as statements and operators. Keywords consist of only lowercase letters and may not be used as a regular identifier such as a variable name. That is, the keyword *if* can not be used as the name of a variable, class, module or any other user-defined identifier. Keywords can be used to begin simple or compound statements and can also act as operators such as the keyword *and*.

2.2 Binding

Like Java, all data in Python is accessed via references. That is, variables, attributes, and items are simply references to data values in memory. Since variables are only references they do not possess a type. The data in which they reference holds the type.

Python variables are only defined when a statement is executed and the variable is bound to a data value. For example, a typical assignment statement would bind the left hand side variable to a string value on the right hand side.

```
hello_string = "Hello World."
```

More precisely, this statement creates an instance of a string object containing the value "Hello World." in which the variable *hello_string* references. Variables that have already been bound can be re-bound to reference a new object. Binding a reference that has already been bound is known as rebinding. Variables can also be de-referenced or unbound which is explained further in Section 2.6.

2.3 Data Types

The Python language represents data values as an object with a given type. The object's type dictates which operations can be performed on the data value and determines the object's attributes, items, and whether or not it can be modified. An object which can be altered is known as a mutable object while an object that can not be changed is referred to as an immutable object.

It was van Rossum's original design decision to simplify the Python language with a small number of powerful data types and little has changed since. Python comes with built-in objects for the following primitive data types: *numbers*, *strings*, *tuples*, *lists*, and *dictionaries*. As with most programming languages user-defined data types known as *classes* can also be constructed.

The *number* objects included in Python support integer (plain and long), floating-point, and complex numbers. *Numbers* are expressed as literals and are immutable objects. This implies that when any operation is applied to a *number* object, a new *number* object is created.

In Python strings, tuples, and lists are all built-in types of an object known as a sequence. "A sequence is an ordered container of items, indexed by non-negative integers." (2) Sequences of other types are also available in the Python library and extension modules. User-defined sequences are also possible.

Python's built-in string object type is simply a sequence of characters typically used to represent text. String objects are immutable and are defined by enclosing a series of characters in single (') or double quotes ("). The traditional backslash can be used to escape literal quotation characters in a string.

```
"This is a string literal."
```

```
'This is also a string literal.'
```

Tuples and lists are two types of sequences that differ by only one semantic quality: mutability. A tuple is an immutable sequence of items (objects) while a list is a mutable sequence of items. Items in tuples and lists can be of the same or differing types. Tuples and lists are defined by comma separated items. Tuples are optionally encapsulated with parentheses and lists with brackets.

```
(7.11, 'hello', 800)      # this is a tuple  
[1, 2.6, 'hi']          # this is a list
```

Python's dictionary type is a mapping between items called keys and their associated values. Although similar to tuples and lists, dictionaries differ greatly in that they are unordered. The almost arbitrarily chosen keys are used to index the list of values. A dictionary is also referred to as an associative array or hash in other languages.

In addition to the primitive types mentioned, Python also presents a null object called *None*. *None* has no methods and no attributes. It is primarily used to define a reference to no object, hence the term "None". Function calls return *None* when no value is explicitly returned.

It is important to note that Python does not support any explicit Boolean type. However, every object can be evaluated as either true or false. That is, an object with a value of zero, *None*, or an empty object is determined to be false. Therefore, empty strings, tuples, lists, and dictionaries are all evaluated to false as well as the values zero or *None*. Every other object is evaluated to true. In Python version 2.2.1 built-in *True* and *False* names were unveiled to support Boolean types and enhance readability. In earlier versions of Python, '1' and '0' were used to represent true and false respectively.

2.4 Typing and Type Checking

Programming languages have different ways of tracking the type of data in which variables reference. In Java and C, the type of variables must be explicitly declared before the variable can be used. Python however, is similar to Perl in that variable typing is performed dynamically during the execution of a program. For example, in the snippet of code below the variable *hello_string* is of type *string* after the first statement is executed because it is referencing an object of type string. However, once the next statement is executed the *hello_string* variable is of type *float* because it is referencing an object of type *float*.

```
hello_string = "Hello World."  
hello_string = 10.1
```

The static typing of Java and C mentioned previously, drastically improves the overall performance of the language by enabling type checking at compile time. Since Python dynamically changes variable type during the execution of a

program, type checking must also be done dynamically during run time. Python offers several built-in functions to check for an object's type. The *type(obj)* function returns the type of the object referenced by the parameter *obj*. The *isinstance(obj,type)* function returns true if the object reference by the *obj* parameter is of the type *type*. In addition, Python supports exception handling by using built-in *try* and *catch* statements similar to Java. Further details concerning exception handling is covered in Section 4.0.

Python also performs coercion for operations and comparisons involving numeric data types. The number with the smaller type is converted to the larger number. For instance, when adding an integer and a floating-point number, the integer number is converted to a floating-point and the sum is stored in a new number of type floating-point.

2.5 Scope

In Python a block is considered a collection of code that is executed as a group. A module, a function body, and a class definition are all considered blocks. The scope in Python defines the visibility of a name within a block.

Local variables defined within a block, have a scope of that block. When a variable is defined within the block of a function, the scope extends to any blocks within the function. However unlike many other programming languages, variables defined within a class block do not extend to the class methods. When a variable is defined at the module level it is known as a global variable. A global variable extends to all functions within the module in which it is defined. Conversely, a local variable is one which is bound to the block in which it is defined.

A Python variable is resolved using the nearest enclosing scope from the block in which it is defined. The set of all visible scopes to a block is known as the block's environment.

2.6 Variable and Object Lifetime

Python provides a statement called *del* which unbinds a variable's reference to an object. Although the *del* statement seems to imply that it removes the object from memory, this is not the case. The *del* statement simply removes the reference to the object. The object is only discarded if no other references to it are present. Discarding objects that no longer have references is known as garbage collection.

The Python garbage collector detects inaccessible objects by counting references to the object. The memory occupied by the objects found to be inaccessible is then freed by the garbage collector.

3.0 Supported Programming Styles

Although most developers only make use of Python's support for object-oriented and procedural programming styles, Python also presents all the features of a powerful functional programming language.

As discussed earlier, Python was originally developed as a scripting language for the Amoeba distributed operating system, so it is no wonder why the procedural programming style is most commonly used. In addition, procedural programming is much easier to learn for the novice programmer than object-oriented or functional programming styles.

Python is considered an object-oriented programming language but unlike Java it also allows developers to write procedural code in the form of modules and functions. In addition, the two programming styles can be used together. For instance, developers can encapsulate data entities with class objects but still use a procedural program to make use of the class object. Unlike most other scripting languages today, Python is a fully object-oriented programming language. It supports class attributes and methods, public and private class variables, inheritance, and overriding.

From the beginning, Python has supported most of the features needed for functional programming. "The basic elements of [functional programming] in Python are the functions *map()*, *reduce()*, and *filter()*, and the operator *lambda*." (4) It is significant that Python implements the *lambda* operator, a distinct feature of functional programming. Currently, Python offers the functional programming features to substitute all control-flow statements (if, for, etc) and implement a completely functional solution.

Since Python is an extendable language, it enables programmers to develop features for the language to support other programming styles. In fact, there is a project in development called Logic-SIG (5) to support logic programming and constraint-propagation features in Python.

4.0 Distinct Features

The Python language is distinct in many ways from most other programming languages in terms of syntax, semantics and built-in language features.

Most programming languages use some sort of encapsulating notation to define a block of code. Java, C and C++ like many other programming languages use left ({) and right (}) braces to define the code block for statements including control flow statements, loops, functions, and classes. Basic and Visual Basic use the *BEGIN* and *END* keywords to denote the beginning and end of a block. Python's creator however, decided to go with an unorthodox and perhaps bold approach to identifying statement blocks. van Rossum simply decided to make use of indentation rather than any beginning and ending notation. The advantage of forcing use of indentation at the language level is that it does not give the developer the freedom to indent code any way they wish, thus making code much more consistent and readable.

Because Python was originally designed for scripting and still often falls into that category today, many find it surprising that it supports exception handling much like

Java. Exception handling can be used to catch and handle run time errors such as division by zero. Since Python uses dynamic type binding, exception handling provides an effective way to manage errors as a result of incompatible types. Not only does Python present a rich set of built-in exception classes, but it allows developers to write custom exception classes as well.

As discussed previously, Python's ease of extendibility is a feature in which some other programming languages lack. That is, some programming languages provide a set of features that can not be complemented by additional code from developers. ABC was van Rossum's motivation for making Python extendable as it did not present the ability to extend the language.

Unlike programming languages like Visual Basic, Python is a portable programming language. That is, Python can be run on any operating system platform where Visual Basic can only be run on Microsoft Windows based systems.

5.0 Domains

Although Python was originally built to accommodate a need for scripting, it has grown into a rich language that can be applied to many programming domains.

Most commonly Python has been used in the Business Applications domain to streamline business processes among other things. Python's productive coding styles and syntax makes for efficient and manageable business applications.

In the Science domain, Python's very high-level data types facilitate management of scientific data and computations. Python has been used to develop successful scientific data visualization tools and mapping systems.

As discussed earlier, Python presents all the necessary features for functional style programming. This style facilitates programming in many domains by minimizing errors and enabling a mathematical approach. The Artificial Intelligence programming domain often makes use of functional programming. In addition, projects like Logic-SIG as mentioned in Section 3.0 are enabling Python to further support programming domains like the Artificial Intelligence domain.

Again due to its easily extendable capability, Python can branch out into other programming domains with relatively little effort.

6.0 Current Status

Today Python is in widespread use all over the world in a variety of applications. However, due to the emergence of a scripting language with striking familiarity to Perl and C, Python is not the premier programming language in use on the web. PHP provides a full featured, cross-platform, efficient, easy to learn alternative to the Perl scripting language. PHP's similarity to Perl and C is undoubtedly why most web application developers have adopted it as their primary development language.

Despite PHP's rise to power, Python is also an increasingly successful programming language being backed by many prestigious companies and organizations such as Google, NASA, Industrial Light & Magic and Philips to name a few. In fact, not only do these companies acknowledge that they are using Python, they strongly believe

that it has contributed to their success. Director of search quality at Google, Inc. says, “Python has been an important part of Google since the beginning, and remains so as the system grows and evolves. Today dozens of Google engineers use Python, and we’re looking for more people with skills in this language.” Mark Shuttleworth of Thawte Consulting says, “Python makes us extremely productive, and makes maintaining a large and rapidly evolving codebase relatively simple.”

Although PHP is a much more popular programming language today, more and more companies and organizations are realizing that Python offers a more productive set of features for enterprise level applications.

7.0 Conclusion

From the beginning, van Rossum had a vision of a simple yet comprehensive language that “adheres to the idea that if a language behaves in a certain way in some contexts, it should ideally work similarly in all contexts.” (2) This concept was the center of an idea which has become a philosophy for many experienced developers today. Of all the praise that Python has received, by far the most common is the increase in development productivity. The increase in productivity can be attributed the clear and easy to maintain code that is produced with the Python language. “Python follows the principle that a language should not have convenient shortcuts, special cases, ad hoc exceptions, overly subtle distinctions, or mysterious and tricky under-the-covers optimizations.” (2)

Python is a powerful general-purpose language that has the support of major players in the software industry. Python’s future is definitely optimistic.

References

1. Venners, Bill. "The Making of Python." [Artima.com](http://www.artima.com/intv/python.html).
<http://www.artima.com/intv/python.html> [accessed 2003-12-09]
2. Martelli, Alex. "Python in a Nutshell." O'Reilly, March 2003.
3. van Rossum, Guido. "Python Reference Manual." [Python.org](http://www.python.org/doc/2.2.3/ref/ref.html).
<http://www.python.org/doc/2.2.3/ref/ref.html> [accessed 2003-12-09]
4. Mertz, David. "Charming Python: Functional programming in Python." [IBM.com](http://www-106.ibm.com/developerworks/library/l-prog.html).
<http://www-106.ibm.com/developerworks/library/l-prog.html> [accessed 2003-12-15]
5. unknown. "Python Logic SIG." [LogiLab.org](http://www.logilab.org/projects/python-logic).
<http://www.logilab.org/projects/python-logic> [accessed 2003-12-15]
6. unknown. "Python Success Stories." [Pythonology.org](http://pythonology.org/success).
<http://pythonology.org/success> [accessed 2003-12-15]

Appendix

Python Grammar in BNF:

```
identifier ::=
    (letter|"_") (letter | digit | "_")*

letter ::=
    lowercase | uppercase

lowercase ::=
    "a..."z"

uppercase ::=
    "A..."Z"

digit ::=
    "0..."9"

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 ASCII character except "\" or newline or the quote>

longstringchar ::=
    <any ASCII character except "\">

escapeseq ::=
    "\" <any ASCII character>

longinteger ::=
    integer ("l" | "L")

integer ::=
    decimalinteger | octinteger | hexinteger

decimalinteger ::=
    nonzerodigit digit* | "0"

octinteger ::=
    "0" octdigit+

hexinteger ::=
    "0" ("x" | "X") hexdigit+

nonzerodigit ::=
    "1..."9"

octdigit ::=
```

```

"0"..."7"

hexdigit ::=
    digit | "a"..."f" | "A"..."F"

floatnumber ::=
    pointfloat | exponentfloat

pointfloat ::=
    [intpart] fraction | intpart "."

exponentfloat ::=
    (intpart | pointfloat)
    exponent

intpart ::=
    digit+

fraction ::=
    "." digit+

exponent ::=
    ("e" | "E") ["+" | "-"] digit+

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

atom ::=
    identifier | literal | enclosure

enclosure ::=
    parenth_form | list_display
    | dict_display | string_conversion

literal ::=
    stringliteral | integer | longinteger
    | floatnumber | imagnumber

parenth_form ::=
    "(" [expression_list] ")"

test ::=
    and_test ( "or" and_test )*
    | lambda_form

testlist ::=
    test ( "," test )* [ "," ]

list_display ::=
    "[" [listmaker] "]"

listmaker ::=
    expression ( list_for
    | ( "," expression )* [ "," ] )

list_iter ::=
    list_for | list_if

list_for ::=
    "for" expression_list "in" testlist
    [list_iter]

list_if ::=
    "if" test [list_iter]

dict_display ::=
    "{ " [key_datum_list] "}"

key_datum_list ::=
    key_datum ( "," key_datum )* [ "," ]

key_datum ::=

```

```

        expression ":" expression

string_conversion ::=
    "`" expression_list "`"

primary ::=
    atom | attributeref
        | subscription | slicing | call

attributeref ::=
    primary "." identifier

subscription ::=
    primary "[" expression_list "]"

slicing ::=
    simple_slicing | extended_slicing

simple_slicing ::=
    primary "[" short_slice "]"

extended_slicing ::=
    primary "[" slice_list "]"

slice_list ::=
    slice_item ("," slice_item)* [","]

slice_item ::=
    expression | proper_slice | ellipsis

proper_slice ::=
    short_slice | long_slice

short_slice ::=
    [lower_bound] ":" [upper_bound]

long_slice ::=
    short_slice ":" [stride]

lower_bound ::=
    expression

upper_bound ::=
    expression

stride ::=
    expression

ellipsis ::=
    "..."

call ::=
    primary "(" [argument_list [","]] ")"

argument_list ::=
    positional_arguments ["," keyword_arguments]
        ["," "*" expression]
        ["," "***" expression]
    | keyword_arguments ["," "*" expression]
        ["," "***" expression]
    | "*" expression ["," "*" expression]
    | "***" expression

positional_arguments ::=
    expression ("," expression)*

keyword_arguments ::=
    keyword_item ("," keyword_item)*

keyword_item ::=
    identifier "=" expression

```

```

power ::=
    primary ["**" u_expr]

u_expr ::=
    power | "-" u_expr
    | "+" u_expr | "\~" u_expr

m_expr ::=
    u_expr | m_expr "*" u_expr
    | m_expr "/" u_expr
    | m_expr "/" u_expr
    | m_expr "\%" u_expr

a_expr ::=
    m_expr | a_expr "+" m_expr
    | a_expr "-" m_expr

shift_expr ::=
    a_expr
    | shift_expr ( "<<" | ">>" ) a_expr

and_expr ::=
    shift_expr | and_expr "&";SPMamp;" shift_expr

xor_expr ::=
    and_expr | xor_expr "\textasciicircum" and_expr

or_expr ::=
    xor_expr | or_expr "|" xor_expr

comparison ::=
    or_expr ( comp_operator or_expr ) *

comp_operator ::=
    "<" | ">" | "==" | ">=" | "<=" | "<>" | "!="
    | "is" ["not"] | ["not"] "in"

expression ::=
    or_test | lambda_form

or_test ::=
    and_test | or_test "or" and_test

and_test ::=
    not_test | and_test "and" not_test

not_test ::=
    comparison | "not" not_test

lambda_form ::=
    "lambda" [parameter_list]: expression

expression_list ::=
    expression ( "," expression ) * [","]

simple_stmt ::= expression_stmt
    | assert_stmt
    | assignment_stmt
    | augmented_assignment_stmt
    | pass_stmt
    | del_stmt
    | print_stmt
    | return_stmt
    | yield_stmt
    | raise_stmt
    | break_stmt
    | continue_stmt
    | import_stmt
    | global_stmt
    | exec_stmt

```

```

expression_stmt ::=
    expression_list

assert_stmt ::=
    "assert" expression ["," expression]

assignment_stmt ::=
    (target_list "=")+ expression_list

target_list ::=
    target ("," target)* [","]

target ::=
    identifier
    | "(" target_list ")"
    | "[" target_list "]"
    | attributeref
    | subscription
    | slicing

augmented_assignment_stmt ::=
    target augop expression_list

augop ::=
    "+=" | "-=" | "*=" | "/=" | "%=" | "**="
    | ">>=" | "<<=" | "&=" | "\textasciicircum=" | "|="

pass_stmt ::=
    "pass"

del_stmt ::=
    "del" target_list

print_stmt ::=
    "print" ( \optionalexpression ("," expression)* \optional","
    | ">\code>" expression
    \optional("," expression)+ \optional"," )

return_stmt ::=
    "return" [expression_list]

yield_stmt ::=
    "yield" expression_list

raise_stmt ::=
    "raise" [expression ["," expression
    ["," expression]]]

break_stmt ::=
    "break"

continue_stmt ::=
    "continue"

import_stmt ::=
    "import" module ["as" name]
    ( "," module ["as" name] )*
    | "from" module "import" identifier
    ["as" name]
    ( "," identifier ["as" name] )*
    | "from" module "import" "*"

module ::=
    (identifier ".")* identifier

global_stmt ::=
    "global" identifier ("," identifier)*

exec_stmt ::=
    "exec" expression

```

```

        ["in" expression ["," expression]]

compound_stmt ::=
    if_stmt
    | while_stmt
    | for_stmt
    | try_stmt
    | funcdef
    | classdef

suite ::=
    stmt_list NEWLINE
    | NEWLINE INDENT statement+ DEDENT

statement ::=
    stmt_list NEWLINE | compound_stmt

stmt_list ::=
    simple_stmt (";" simple_stmt)* [";"]

if_stmt ::=
    "if" expression ":" suite
    ( "elif" expression ":" suite )*
    ["else" ":" suite]

while_stmt ::=
    "while" expression ":" suite
    ["else" ":" suite]

for_stmt ::=
    "for" target_list "in" expression_list
    ":" suite
    ["else" ":" suite]

try_stmt ::=
    try_exc_stmt | try_fin_stmt

try_exc_stmt ::=
    "try" ":" suite
    ("except" [expression
               ["," target]] ":" suite)+
    ["else" ":" suite]

try_fin_stmt ::=
    "try" ":" suite
    "finally" ":" suite

funcdef ::=
    "def" funcname "(" [parameter_list] ")"
    ":" suite

parameter_list ::=
    (defparameter ",")*
    ("*" identifier [, "*" identifier]
     | "*" identifier
     | defparameter [","])

defparameter ::=
    parameter ["=" expression]

sublist ::=
    parameter ("," parameter)* [","]

parameter ::=
    identifier | "(" sublist ")"

funcname ::=
    identifier

classdef ::=
    "class" classname [inheritance] ":"

```

```
        suite
inheritance ::=
    "(" [expression_list] ")"
classname ::=
    identifier
file_input ::=
    (NEWLINE | statement)*
interactive_input ::=
    [stmt_list] NEWLINE | compound_stmt NEWLINE
eval_input ::=
    expression_list NEWLINE*
input_input ::=
    expression_list NEWLINE
```