Basic Syntax

|  |  |  |
| --- | --- | --- |
| **Feature** | **Python** | **B4P** |
| **Basics** | Interpreted: Yes (Similar to PERL, PHP)  Interactive: Yes  Object-Oriented: Yes  General pupose language with very large collection of libraries and community | Interpreted: Yes (Precompiles into tokenized structure)  Interactive: Yes  Object-Oriented: No  Powerful special-purpose language focusing on large scale data processing around tables |
| **Overall Language Syntax** | Derived from languages like MODULA (Pascal successor), Algol 68, Smalltalk, UNIX shells, and less from C/C++ | C /C++ / Java style syntax |
| **Reserved words** | **and** **exec** **not**  **assert** **finally** **or**  **break** **for** **pass**  **class** **from** **print**  **continue** **global** raise  **def** **if** **return**  **del** **import** try  **elif** **in** **while**  **else** **is** with  **except** **lambda** yield | true false else  new line tab escape  The main language keywords like **if**, **for**, etc. are treated as B4P function names. You can actually create variables and tables with names like **if** and **for**. |
| **Code blocks** | Realized with indentations where the number of characters used for indentation is relevant. Same number of spaces for one block  if b > a:  print("b is greater ")  print("than a")  else:  print("a is greater than b") | Realized in C/C++ style  if (b>a)  {  echo( "b is greater" );  echo( "than a ");  }  else: echo( "a is greater than b" );  // Colon after else is required if not followed by another block { } |
| **Multiple Statement Groups as Suites (Blocks)** | Python defines code blocks containing multiple statements as **suites**, beginning with **header lines** containing if, for, etc.  if expression :  suite (placeholder for statements)  elif expression :  suite  else :  suite | Realized in C/C++ style  if (expression)  statement;  else: if (expression)  statement;  else:  statement  Or:  if (expression)  {  multiple statements;  }  else: if (expression)  {  multiple statements  }  else // No colon after else if beginning with {  {  multiple statements  } |
| **Multi-Line Statements** | Previous line must end with backslash | Like C/C++. No dedicated symbols needed. Statements end with semicolon at the end. Only literal (string) constants must end on the same line, closing with quotation marks. |
| **Multiple statements in single line** | Separate the two statements with semicolon | Same. |
| **Quotations** | Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.  The triple quotes are used to span the string across multiple lines. For example, all the following are legal | B4P accepts single and double quotes, but provide provide different attributions to the strings. Single quoted literals permit more flexible interpretation of the contents (e.g. wildcards, interpreteation of comma-separated contents inside, etc.) where double quoted literals require contents to be treated as they are.  See: **quoted literals**, **softquoted literals**. |
| **Comments** | Somehow a UNIX style  # single line comment  ...  Multi-line  comments  ... | C/C++ style:  // single line comment  /\*  Multi-line  comments  \*/  B4P code inside github markdown files (.md) are considered as comments except in designated code blocks using three accent-grave symbols.  Some documentary contents  ```  echo("B4P code");  ```  Documentary contents follow |
| **Waiting for user** | raw\_input("\n\nPress the enter key to exit.") | input quick( "Press enter key to exit", new line );  pause; |
| **Multiple assignments** | Nice features.  a = b = c = 1  a, b, c = 1, 2, "Hello" | Not supported for variables.  But multiple assignments into multiple cells in a table is supported:  [ table: Name, .. ] = { Chris, Nic, Claus }; |

Variables

|  |  |  |
| --- | --- | --- |
| **Feature** | **Python** | **B4P** |
| **Explicit Declaration** | Not needed  Declaration happens at first assignment | Not needed  Declaration happens at 1st assignment |
| **Variable names** | Single word, few special characters, no spaces allowed.  All variable names are case sensitive. | Full freedom, e.g. spaces allowed. Without quotation marks, multiple consecutive spaces inside are treated as one space. True variable names to be used (containing multiple spaces and even reserved symbols like +, -, \* ) need to be put into quotation marks.  All variable names are case sensitive.  All variables require [ ] at the end. |
| **Deleting variables** | del var1  del var1, var2; | delete( var1[], var2[] ); |
| **Variable Scopes** | global  local (visible inside a function only) | global  local  system System variables |
| **Global Variables** | All variables defined outside functions.  Inside functions, global variables can be declared with 'global' | All variables defined outside functions **and**  all variables defined in B4P programs started by parent B4P programs using **start**(…) or **include**(…).  Inside functions (and called B4P programs) global variables can be declared with the **global** function. |
| **Basic Data Types** | strings  numbers (different types, see below)  list  tuple  dictionary  Additional data types:  range, dict (dictionary)  bool (Values: True, False)  Binary types: byte, bytearray, memoryview | literal (True UNICODE, and every character counts 1)  numeral  parameter set  boolean (Values: true, false)  date (with and without time or time only)  void  Ranges: Use parameter sets.  Binary values: Not directly supported. Some B4P programs use hexadecimal contents in strings. |
| **Numbers** | int  long (integers of unlimited size, apparently obsolete)  float  complex | Only one type: **numeral**  Internally treated as double precision floating point, but no issues using them as whole numbers, e.g. loop index. |
| **Strings** | str = 'Hello World!'  print str # Prints complete string  print str[0] # Prints first character of the string  print str[2:5] # Prints characters starting from 3rd to 5th  print str[2:] # Prints string starting from 3rd character  print str \* 2 # Prints string two times  print str + "TEST" # Prints concatenated string | str[]= "Hello World!";  echo( str[] );  echo( str[]{0} );  echo( str[]{2..5} );  echo( str[]{2..-1} );  // String multiplication not supported in B4P  echo( str[] + "TEST" );  Combined arithmetics of strings and numbers are restricted intentinally, but arithmetics of numbers and blank strings ('') are allowed where blank strings are treated as zero (but '0' is not). |
| **Python: Lists**  **B4P: Arrays** | Lists are compound variables are are treated as arrays (like in C/C++).   * Ordering of the values is retained * Duplicates are allowed   Pythone syntax permits accessing the members as if they are elements of a set.  list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]  tinylist = [123, 'john']  print list # Prints complete list  print list[0] # Prints first element of the list  print list[1:3] # Prints elements starting from 2nd till 3rd  print list[2:] # Prints elements starting from 3rd element  print tinylist \* 2 # Prints list two times  print list + tinylist # Prints concatenated lists  List elements can be deleted with **del**.  Iteration:  for x in l: print x | In B4P, these are arrays which are also compound variables containing unnamed members. Every member is treated as a separate variable. The B4P function **set** ( … ) puts all array members into a parameter set value.  array( list[], { 'abcd', 786 , 2.23, 'john', 70.2 } )  array( tinylist[], {123, 'john'} )  echo( set(list[]) )  echo( set(list[]){0} )  echo( set(list[]){1..3} )  echo( set(list[]){2..-1} )  list[] +<== tinylist[] // Combine both arrays  echo( set(list[]) )  In B4P, arrays are defined with dim or array or similar functions, and arrays can also be created as ad hoc. Arrays can be resized at any time.  Array members can be deleted with the **delete**(…) function  Iteration:  for all variables( l[], x[]) echo( x[] ); |
| **Python: Tuples**  **B4P: Protected Arrays** | Similar to lists, but contents and size cannot be changed. Consider them as 'read-only' lists after created. Syntax with parentheses.  list = ( 'abcd', 786 , 2.23, 'john', 70.2 ) | Not supported. If you need read-only protection of arrays , then use **dim protect** or **array protect**. |
| **Python: Dictionary**  **B4P: Structures** | Like lists, dictionaries are compound variables which are referenced by their names and not by index values.  dict = {}  dict['one'] = "This is one"  dict[2] = "This is two"  tinydict = {'name': 'john','code':6734, 'dept': 'sales'}  print dict['one'] # Prints value for 'one' key  print dict[2] # Prints value for 2 key  print tinydict # Prints complete dictionary  print tinydict.keys() # Prints all the keys  print tinydict.values() # Prints all the values  Deleting and clearing:  del dict['Name'] # remove entry with key 'Name'  dict.clear(); # remove all entries in dict  del dict ; # delete entire dictionary | B4P supports structures which are compound variables with named members.  dict[one] = "This is one"; // Quotation marks not required  dict['2'] = "This is two"; // Required here, 2 as literal  structure( tinydict[],{ name,code,dept }, { john,6734,sales});  echo (dict[one]);  echo (dict['2']);  inspect( tinydict[] );  echo( set names( tinydict[]) );  echo( set (tinydict[]) );  Deleting and clearing:  delete( dict[Name]);  delete members( dict[] );  delete ( dict[] ); |
| **Python: Sets**  **B4P: Parameter Sets** | Similar to lists, but following differences apply:   * Values are unordered * Duplicates are forbidden. * Set operations (intersection, union, etc.) are supported * Items in the set cannot be changed once created   Syntax:  a\_set = { 'abcd', 786 , 2.23, 'john', 70.2 } | B4P parameter sets are not compound variables, but treated as simple variables containing structured information.   * Values are ordered * Duplicates are allowed, but can be eliminated with **trim**(). * Set operations (intersection, union, etc.) are supported * Items can be changed, extended or reduced   a set[] = { 'abcd', 786 , 2.23, 'john', 70.2 }; |
| **Frozen sets** | frozen = frozenset( a\_set ) | In B4P, apply protection settings of choice, e.g.  protect(read only, a[ ] ); |
| **Ranges** | x = range(6) # x is of range class  for n in x:  print(n) | Use B4P parameter sets to define ranges in similar way  x[] = { 0..5 };  for all parameters( x[], n[] ) echo( n[] ); |
| **Variable Type conversions** | type( a )  int ( a ), long ( a ), float ( a )  complex ( r, i)  str ( a )  len ( x ) | type( a[] )  numeral( a[] )  // Complex numbers not supported in B4P  literal ( a[] )  member count( x[] ) // for arrays, structures  length ( x[] ) or x[]{} // for parameter sets |
| **repr**  **B4P: expression** | var = 'foo'  print(repr(var))  x = 1  print(eval('x + 1')) | var[] = foo;  echo( expression( :'var[]' ) ); // As string  echo( expression( var[] ) ); // As code  x[] = 1;  echo( expression( :'x[]+1') ); // As string  echo( expression( x[]+1) ); // As code |
| **Conversion to tuple** | tuple ( "a,b,c,d,e" ) | parameter set( "a,b,c,d,e") |
| **Conversion to list** | a = ('a', 'b', c', 'd', 'e')  print(list(vowel\_tuple))  Python also converts single letters to elements  a = 'aeiou'  print(list(vowel\_tuple)) | array( a[], parameter set( "a,b,c,d,e" ) );  inspect(a[]);  The scheme of using single letters (like aeiou) is not implemented in B4P. |
| **Conversion to sets** | # empty set  print(set())  print(set('Python')) # from string  print(set(('a', 'e', 'i', 'o', 'u'))) # from tuple  print(set(['a', 'e', 'i', 'o', 'u'])) # from list  print(set(range(5))) # from range | echo( parameter set ('') )  array( a[], { P, y, t, h, o, n } ); // Not reordered  array( b[], +{ P, y, t, h, o, n } ); // Reordered  inspect( a[] );  inspect( b[] );  structure( a[], { Name, Score }, { Jane, 25 } );  b[] = set( a[] ); |
| **Python: Modules**  **B4P: B4P Programs to include** | Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.  Use **import** (…) to import modules.  Python also supports importing individual functions and attributes from modules rather than everything, e.g.  from fib import fibonacci  From the file "fib.py", import the function "Fibonacci" only. | Create an own B4P library file containing all the definitions you need, save it and call it up with the **include** (…) call.  Selective importing is not possible, but you can provide additional function parameters into the **include**(…) call which is then available as command line parameter inside that module and can be used to include specific features (e.g. function definitions) or not.  **include**( My Library.b4p, essential functions only ); |
| **dir() Function** | Generates a list of all function names found in the module imported  content = dir(math)  print content | B4P: Not supported |
| **globals() and Locals() Function** | Gathers names of all global and local variables | For inspection, following functions are available   * list variables (lists all variables) * list global variables * list local variables * list system variables |
| **reload() Function** | When a module is imported, then the top level code (outside function definitions) is executed only once, even if imported repeatedly.  With **reload**() you can force re-execution | Different philosophy, execution is ruled internally. In a B4P library file, put all code into a **once** block:  once  {  define fuction … // All the code is executed once  }  echo( Done ); // This code is executed at every include.  If you want to influence this, then provide additional parameters into the include statement and use them in the library to decide which code to execute and which not. |
| **References** | If a variable is assigned to another variable, e.g. **a = b**, then a is actually a reference to b, without copying the values. Once a different value is assigned to a, then a new object will be created. | Assignment in B4P does copy the actual values.  However, references can be defined explicitly, e.g.  **a[] =^ b[];** |

Operators

|  |  |  |
| --- | --- | --- |
| **Feature** | **Python** | **B4P** |
| **Arithmetic Operators** | +, -, \*. /  % (Modulo)  \*\* (Exponent)  // (Floor division) | +, -, \*. /  =mod (…)  =exp( …, … )  = round down ( … / … ) |
| **Comparison Operators** | ==, !=  <, <= ,>, >= | =, <> These operators accept more than 1 operand on the right-hand side, e.g. a[] = 2,3,5 or a[] = 3..5  <, <= ,>, >=  ==, !=: These are strict operators, accepting 1 operand only |
| **Assignment Operators** | +=, -=, \*=, /=,  %=, \*\*=, //= | +=, -=, \*=, /= |
| **Deep Operators** | Not supported | ^ suffix added to operators, to apply operators on parameter set members and not on the main values.  +^, -^, etc. +^^, -^^, etc. |
| **Bitwise Operators** | &, |, ^, ~, <<, >> | Not supported in B4P for reason that bitwise operations are very unlikely in typical B4P applications |
| **Logical Operators** | and or not | & | ! |
| **Membership Operators** | a **in** b returning True or False (alternative: not in) | Structures: existing( a[x] )  Parameter sets: find( a[ ], b[ ] ) returning position or -1 |
| **Identity** | a **is** b (alternative: a is not b)  True if a and b refer to the same memor location | Not supported. |
| **Operators Precedence** | Appears pragmatic for available operators | Appears pragmatic for available operators |
| **Operations on strings** | + (catenation)  \* (repetition)  [ ] (slice) | + (catenation)  \* (Not supported, but blank literal and numeral combined with binary operator will treat blank literal as zero)  { } (picking details, similar to slice in Python)  - operator (remove one occurrence of a string section)  / operator (remove all occurrences of a string section) |
| **Escape Characters** | Python supports various escape chars like in C, C++; Java  \t, \n  \b (backspace) | B4P uses a different approach  tab, new line  &#08; (Use HTML entity in softquoted literal for backspace) |
| **Raw String** | Prefix **r** before the string  r'raw string' | Define a **quoted literal**:  "raw string" |
| **String Formatting** | %d, etc. (like in printf in C/C++)  print "My name is %s and weight is %d kg!" % ('Zara', 21) | Not supported in C/C++/Java/Python style. use **literal** (…) to convert variables to strings with given formatting. |
| **UNICODE** | Normal strings in Python are 8-bit characters.  UNICODE strings are 16-bit characters. | B4P uses UTF-8 transparently internally, which requires 1 byte for traditional ANSI characters, and 2-4 bytes for all other non-ANSI characters even beyond the *Basic Multilingual Plane*.  Every character counts as 1 character, even non-ANSI characters. |

Flow Control

|  |  |  |
| --- | --- | --- |
| **Feature** | **Python** | **B4P** |
| **if**  **B4P: if, unless** | var = 100  if ( var == 100 ) : print "Value of expression is 100" | var[] = 100;  if ( var[] = 100 ) echo( Value of expression is 10 );  unless ( var[] <= 5 ) echo( Value is greater than 5 ); |
| **Loops** | While loops  For loops  Loop nesting |  |
| **Influencing loops** | break  continue  pass | break (or break loop)  continue (or continue loop)  placeholder statement. Use **null;** |
|  |  |  |

Date and Time

|  |  |  |
| --- | --- | --- |
| **Feature** | **Python** | **B4P** |
| **Basics** | Implemented in the **time** module which is a library to add.  Time intervals are floating point values, 1.0 = 1 second, beginning in Jan 1, 1970 and ending in 2038.  A time-tuple is provided with 9 values: year, month, day, hour, minute, second, day of week, day of year, daylight saving time | B4P provides a dedicated date variable type which can store   * Just the date without time * Just the time without date * Date and time * Blank dates   Time intervals are numerals, 1.0 = 1 day. (1/24) is 1 hour.  Arithmetics: Differences between two dates give numerals. Numerals can be added / subtracted to/from dates to change days and time.  The function **reschedule**(…) is a powerful function to modify dates. Absolute dates, weekdays, day of year, etc. can be retrieved with various B4P functions. |
| **Get current time** | localtime = time.localtime(time.time())  print "Local current time :", localtime | localtime[] = date ( now );  echo( "Local current time: ", localtime[] ); |
| **Sleep for specified time** | Supported | Supported |
| **Outputting calendar** | Python provides a nice feature to print a calendar as multi-line string for a chosen month | Not supported, but easy to program in B4P |
|  |  |  |

Defining User Functions

|  |  |  |
| --- | --- | --- |
| **Feature** | **Python** | **B4P** |
| **Basics** | Python comes along with a 'docstring'  def printme( str ):  "This prints a passed string into this function"  print str  return;  # Now you can call printme function  printme("I'm first call to user defined function!")  printme("Again second call to the same function") | Define procedure (without ret value) or define function (with ret value)  define procedure( print me, {{ str, literal }} )  // This prints a passed string into this function  {  echo( str[] );  }  printme("I'm first call to user defined function!");  printme("Again second call to the same function"); |
| **Pass by Reference vs. Value** | All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example | In B4P, you can pass parameters as   * Input value (default) * Output value (no need to pre-define before calling) * I/O value * References (like Python) |
| **Function Arguments**  **B4P: Function Parameters** | Following types of formal arguments exist:   * Required arguments * Keyword arguments * Default arguments * Variable-length arguments |  |
| **Keyword Arguments** | printme( str = "My string") | Not supported in B4P |
| **Default Arguments** | # Function definition is here  def printinfo( name, age = 35 ):  "This prints a passed info into this function"  print "Name: ", name  print "Age ", age  return;  # Now you can call printinfo function  printinfo( age=50, name="miki" )  printinfo( name="miki" ) | // Function definition is here  define procedure( print info,  {{name, literal},{age, numeral}},1,2 )  {  if (parameter count[]<2) age[] = 35;  echo( "Name: ", name[] );  echo( "Age ", age[] );  }  // Now you can call printinfo function  print info( miki, 50 );  print info( miki ); |
| **Variable Length Arguments** | 1st parameter provided in arg1, remaining ones via \*vartuple  # Function definition is here  def printinfo( arg1, \*vartuple ):  "This prints a variable passed arguments"  print "Output is: "  print arg1  for var in vartuple:  print var  return;  # Now you can call printinfo function  printinfo( 10 )  printinfo( 70, 60, 50 ) | Excess parameters not named in function definition are called  **parameter n[]**. Local variable **parameter count[]** is also available.  // Function definition is here  define procedure( print info, {{parameter 1, numeral}},1,-1 )  {  echo( "Output is: " );  for (i[] = 1, i[] <= parameter count[], i[]++ )  echo( ("parameter " + literal(i[] ))[] );  }  // Now you can call printinfo function  print info( 10 );  print info( 70, 60, 50 ); |
| **Lambda Functions** | Allows to define simple functions in one single line  sum = lambda arg1, arg2: arg1 + arg2;  # Now you can call sum as a function  print "Value of total : ", sum( 10, 20 ) | Use **define function**(…) to define functions providing return values  define function( my sum, {{n1, numeral},{n2, numeral}} )  { return( n1[]+n2[]); }  // Now you can call printinfo function  echo( my sum( 10, 20 )); |
| **Return statement** | Supported | Supported. Only required when returning values. |
| **Yield statement**  **B4P: Use once()** | # A simple generator function  def my\_gen():  n = 1  print('This is printed first')  # Generator function contains yield statements  yield n  n += 1  print('This is printed second')  yield n  n += 1  print('This is printed at last')  yield n  my\_gen()  my\_gen()  my\_gen() | // A simple generator function  define procedure( my gen )  {  once() echo('This is printed first');  else: once() echo('This is printed second');  else: echo('This is printed at last');  }  my gen;  my gen;  my gen; |

File I/O

|  |  |  |
| --- | --- | --- |
| **Feature** | **Python** | **B4P** |
| **Basics** | Similar to other traditional languages, Python provides functions to open files, read and/or write contents and close them again.  A file object will be created, with methods available to read, write, etc. | Load text files entirely using **table load** (file name, TEXT), then process them in a table, and finally do **table save** ( file name, TEXT ) to save the work. |
|  |  |  |

Managing Directories and Files

|  |  |  |
| --- | --- | --- |
| **Feature** | **Python** | **B4P** |
| **Basics** | The **os** module contains various functions. | Integral part of B4P |
| **Rename files** | os.rename( "test1.txt", "test2.txt" ) | file rename ( "test1.txt", "test2.txt" ); |
| **Remove files** | os.remove( "test1.txt" ) | file delete ( "test1.txt" ); |
| **Create directories** | os.mkdir( "newdir " ) | directory create ( newdir ); |
| **Change directory** | os.chdir( "newdir " ) | working directory ( newdir ); |
| **Get working directory** | os.getcwd() | working directory () (called as function) |
| **Remove directory** | os.rmdir ( "newdir" ) | directory delete ( newdir ); |
| **Create symbolic link** | Not supported | file create link ( link name, target directory ); |
|  |  |  |

Exceptions

|  |  |  |
| --- | --- | --- |
| **Feature** | **Python** | **B4P** |
| **Assert Exceptions** | assert Expression[, Arguments] | B4P provides following function assert exception and stop execution.  exception ( … ) |
| **Exception Handling** | try:  You do your operations here;  ......................  except ExceptionI:  If there is ExceptionI, then execute this block.  except ExceptionII:  If there is ExceptionII, then execute this block.  ......................  else:  If there is no exception then execute this block. | B4P needs no **try** statement. In case of exeptions, consider throwing.  throw ( exception 1 )  . . .  catch if( a[], exception 1 )  {  . . . .  }  catch if( a[], exception 2 )  {  . . . .  } |
| **Finally** | Supported | Not supported directly, but possible with catch – throw again sequence:  Recommend to catch values thrown, execute some code, and continue throwing |
| **User-Defined Exceptions** | Supported | Not supported |

Object Oriented Programming (Classes, Object Instances)

|  |  |  |
| --- | --- | --- |
| **Feature** | **Python** | **B4P** |
| **OOP** | Supported | Not supported |