

# **The Pocket IGCSE Pseudocode to Python** **Reference Guide**

**Eason Qin Luojia** (eason@ezntek.com)

*Sixth Revision*

*December 1<sup>st</sup>, 2024*

## Note 1

For my classmates and fellow G1/G2 Computer Science Students, I **EXPECT** you to have read this document prior to reading the next few pages. **PLEASE DO NOT** ask me questions that have information contained in any of these notes. I will refuse to answer your questions until you have clearly read every page of this document.

I **EXPECT** you to know that this document is just a simple side-by-side comparison/reference as to the differences between IGCSE Pseudocode and Python. I **EXPECT** you to know that this is **NOT COMPREHENSIVE!** This **does not cover and does not intend to teach HOW** to program in pseudocode! I will be releasing a guide as to how to program in Pseudocode when the time comes. *If the guide is already out, please head to <https://ezntek.com/revision> to find it.*

## Note 2

All values in angle brackets, like so:

```
<variable name>  
<type>  
<value>
```

represent *meta-variables* or *meta-values*, which should wholly, i.e. including the beginning angle bracket, <, to the ending angle bracket, >, be replaced with an actual value that is described within the brackets.

**In layman's terms**, everything between <> should be replaced with what it *says* inside. You should not write the <> either.

## Note 3

If there is an item that leaks onto a new line, such as,

```
FOR <counter> ← <begin> TO <end> STEP  
  <step>  
  <statement>...  
NEXT <counter>
```

Count it as if it were equal to:

```
FOR <counter> ← <begin> TO <end> STEP <step>  
  <statement>...  
NEXT <counter>
```

## Note 4

Some key definitions will be made:

Term	Meaning
Expression <expr>	Any <b>variable name or value, function calls, or arithmetic expressions, enclosed or not enclosed in brackets</b> . It <b>will</b> be shortened to <b>expr</b> when necessary.
Identifier <ident>	A <b>variable name</b> . It <b>will</b> be shortened to <b>ident</b> when necessary.
Operator	a symbol that does something, such as math. They include symbols such as * + - / etc.
...	Represents repetition, i.e. repeated statements. If there is a comma, such as  <statement>, ...  That implies that there can either be one statement <statement>, or many statements separated by a comma, such as  <statement>, <statement>, <statement>

## Note 5

This work copyright © Eason Qin 2024, and is licensed under the Creative Commons Attribution-ShareAlike-NonCommercial 4.0 International license. Visit the Creative Commons Website (<https://creativecommons.org>) for details. **All prior versions are also licensed under the same license.**

## Note 6

This is the **fourth revision** of the guide. If you have earlier revisions, view the changelog:

1. **Initial version.**
2. **Fixed syntax highlighting** added consistency in the *Functions* section, and added this note.
3. **Added a License.**
4. **Fixed inconsistencies in the notes**, and slight syntax highlighting changes
5. **Fixed critical error in the For loop section**
6. **Fixed if statement indentations**

## Reference Guide

Item	IGCSE Pseudocode	Python
<u>Comment</u> <i>Used to annotate code.</i>	<pre>// This is a comment. // To comment, simply put two // slashes (//) in front of your text.</pre>	<pre># This is a comment. # To comment, simply put one # hashtag (#) in front of your # text.</pre>
<u>Values</u> <i>Also known as Literals, they represent values.</i>	<pre>// These are all INTEGER's, or <b>whole</b> // <b>numbers</b> <b>42</b> <b>-2043</b>  // These are all REAL's, or <b>decimal</b> // <b>numbers</b> <b>3.14159</b> <b>2.718282</b> <b>56.52</b>  // These are STRING's, or "text" // (enclosed in only "): "Good morning, user!" "Thomas" "Jason Lee"  // These are BOOLEAN's, either TRUE or FALSE <b>TRUE</b> <b>FALSE</b>  // These are CHAR's, or singular // characters (enclosed only in '): 'c' 'F' 'b'</pre>	<pre># These are all int's, or <b>whole</b> # <b>numbers</b> <b>42</b> <b>-2043</b>  # these are all float's, or <b>decimal</b> # <b>numbers</b> <b>3.14159</b> <b>2.718282</b> <b>56.52</b>  # These are str's, or "text" # (enclosed in both " and ') "Good morning, user!" 'Thomas' 'Jason Lee'  # These are bool's, either TRUE or # FALSE <b>True</b> <b>False</b>  # there is no CHAR in Python, just use a str.</pre>
<u>Declaring a variable</u> <i>This is to make it clear to the computer that the variable exists.</i>  <i>This is not necessary in Python.</i>	<pre><b>DECLARE</b> &lt;variable name&gt;: &lt;type&gt;  // e.g. <b>DECLARE</b> Name: <b>STRING</b> <b>DECLARE</b> TotalScore: <b>INTEGER</b> // or, <b>DECLARE</b> Name:<b>STRING</b> <b>DECLARE</b> TotalScore:<b>INTEGER</b></pre>	<pre>&lt;variable name&gt;: &lt;type&gt;  # e.g. name: <b>str</b> total_score: <b>int</b></pre>

<u>Assignment</u> <i>This is used to give a value to a previously declared variable.</i>	<pre>&lt;variable name&gt; ← &lt;expression&gt; // NOTE: you may write it like &lt;- in // your computer.  // e.g. Name ← "Thomas" TotalScore ← 84 Name ← FirstName</pre>	<pre>&lt;variable name&gt; = &lt;expression&gt;  # e.g. name = "Thomas" total_score = 84 name = first_name</pre>
<u>Input and Output</u> <i>This is used to give users feedback and receive input.</i>	<pre><b>OUTPUT</b> &lt;expression&gt; <b>OUTPUT</b> &lt;expression&gt;, ... // Print however many things you // require.  <b>INPUT</b> &lt;expression&gt;  // e.g. <b>OUTPUT</b> "What is your name" <b>OUTPUT</b> "Welcome", Name <b>OUTPUT</b> "What is your Social Security Number?" <b>INPUT</b> SocialSecurityNumber <b>OUTPUT</b> "What is your ID?" <b>INPUT</b> ID</pre>	<pre><b>print</b>(&lt;expression&gt;) <b>print</b>(&lt;expression&gt;, ...) # Print however many things you # require.  &lt;variable name&gt; = <b>input</b>(&lt;prompt&gt;)  # e.g. <b>print</b>("What is your name") <b>print</b>("Welcome", name)  # Note that if you need to input # something into an integer, you must # wrap input in int, or separate them # like so: social_security_number = <b>int</b>(<b>input</b>()) id = <b>input</b>("What is your ID?") id = <b>int</b>(id)</pre>
<u>Arithmetic (expression)</u> <i>This is to do math.</i>	<pre>&lt;expr&gt; &lt;operator&gt; &lt;expr&gt;  // e.g. 2 + 5 (3 * X) + 1  // you can combine it with an // assignment, like so: NextTerm ← X + 1</pre>	<pre>&lt;expr&gt; &lt;operator&gt; &lt;expr&gt;  # e.g. 2 + 5 (3 * x) + 1  # you can combine it with an # assignment, like so: next_term = x + 1</pre>
<u>Arithmetic Assignments</u> <i>This is to perform a math operation on the variable itself, including incrementing a variable, etc.</i>	<pre>// They DO NOT exist in pseudocode, // but may be substituted with:  &lt;ident&gt; ← &lt;ident&gt; &lt;operator&gt; &lt;expr&gt;  // e.g. Age ← Age + 1 Temperature ← Temperature - 5</pre>	<pre>&lt;ident&gt; &lt;operator&gt;= &lt;expr&gt;  # e.g. age += 1 temperature -= 5</pre>

<u>Comparison Operators</u> <i>This is to check the relation between two values, such as equality, greater or less than, not equal to, etc.</i>	<pre>// Equality Age = 18  // Greater than, less than Age &gt; 18 Age &lt; 18  // Greater than or equal to, less than or equal to Age &gt;= 18 Age &lt;= 18  // Not equal to Age &lt;&gt; 18</pre>	<pre># Equality age == 18  # Greater than, less than age &gt; 18 age &lt; 18  # Greater than or equal to, less than or equal to age &gt;= 18 age &lt;= 18  # Not equal to age != 18</pre>
<u>Boolean Expressions</u> <i>This is akin to logic gates; it is to process one or two boolean values and evaluate it to True or False depending on the operator.</i>	<pre>// is one condition TRUE AND the other one true? ConditionOne AND ConditionTwo  // is one condition TRUE OR the other one true? ConditionOne OR ConditionTwo  // is the condition NOT true? NOT Condition</pre>	<pre># is one condition TRUE AND the other one true? condition_one and condition_two  # is one condition TRUE OR the other one true? condition_one or condition_two  # is the condition NOT true? not condition</pre>
<u>Conditional Branching (if)</u> <i>This is to make a decision, a choice, to ask a question, whichever interpretation pleases you.</i>	<pre>// either: IF &lt;condition&gt;     THEN          // PRESS SPACE TWICE!     &lt;code&gt;         // PRESS SPACE TWICE! ELSE     &lt;code&gt;         // PRESS SPACE TWICE! ENDIF  // or: IF &lt;condition&gt;     THEN     &lt;code&gt; ENDIF  // e.g. IF Age &gt; 18     THEN     OUTPUT "you can drink!" ELSE     OUTPUT "you cannot drink..." ENDIF</pre>	<pre>if &lt;condition&gt;:     &lt;code&gt;    # PRESS SPACE 4 TIMES! else:     &lt;code&gt;  # or if &lt;condition&gt;:     &lt;code&gt;  # e.g. if age &gt; 18:     print("you can drink!") else:     print("you cannot drink...")</pre>

<p><u>Chained conditional branching (if-else if-else)</u>  <i>This is to ask multiple questions in a row.</i></p> <p><b>Note that in pseudocode, you must follow this indentation exactly, i.e. <i>THEN</i> must be on a new line and indented by 2 spaces, and the code block by 4, <i>ELSE</i> by none, and the code block that follows by 2.</b></p> <p><b>ALL OTHER CODE BLOCKS ARE INDENTED BY 4 SPACES.</b></p>	<pre>// This does not exist in pseudocode, // but can be emulated in the following // way: IF &lt;condition&gt;   THEN     &lt;code&gt;   ELSE     IF &lt;condition&gt;       THEN         &lt;code&gt;       ELSE         &lt;code&gt;     ENDIF   ENDIF  // with the IF statement inside the // larger ELSE statement being able // to be repeated as many times as // needed.  IF Age &gt; 18   THEN     OUTPUT "You can drink!"   ELSE     IF Age &gt; 16       THEN         OUTPUT "You can almost drink!"       ELSE         OUTPUT "You can't drink..."     ENDIF   ENDIF</pre>	<pre>if &lt;condition&gt;:     &lt;code&gt; elif &lt;condition&gt;:     &lt;code&gt; else:     &lt;code&gt;  # e.g. if age &gt; 18:     print("you can drink!") elif age &gt; 16:     print("you can almost drink!") else:     print("you can't drink...")</pre>
<p><u>Pattern Matching</u>  <i>This is like finding a value that matches the one that you have, and then doing something when you find it.</i></p> <p><b>NOTE that using match in Python requires version 3.10 or later. If you use the latest version of Thonny or Replit, you will be OK.</b></p>	<pre>CASE OF &lt;expr&gt;   &lt;expr&gt;: &lt;statement&gt;   &lt;expr&gt;: &lt;statement&gt;   ...   // optionally,   OTHERWISE &lt;statement&gt; ENDCASE  // e.g. CASE OF BottleMaterial   "Plastic": OUTPUT "Unsustainable..."   "Metal": OUTPUT "Sustainable!"   "Glass": OUTPUT "Fragile..."   "Paper": OUTPUT "WHY?"   OTHERWISE OUTPUT "Unrecognized" ENDCASE</pre>	<pre>match &lt;expr&gt;:   case &lt;expr&gt;:     &lt;code&gt;   case &lt;expr&gt;:     &lt;code&gt;   ...   # This is equivalent to OTHERWISE   case _:     &lt;code&gt;  match bottle_material:   case "Plastic":     print("Unsustainable...")   case "Metal":     print("Sustainable!")   case "Glass":     print("Fragile...")   case "Paper":     print("WHY?")   case _:     print("Unrecognized")</pre>

<p><u>Pre-condition iteration (while)</u></p> <p><i>This is to repeatedly do tasks, while some condition is true (so to not infinitely loop).</i></p>	<pre> WHILE &lt;condition&gt; DO     &lt;code&gt; ENDWHILE  // e.g. WHILE Number &gt; 1 DO     Number ← Number - 1     OUTPUT "The number is now", Number ENDWHILE </pre>	<pre> while &lt;condition&gt;:     &lt;code&gt;  # e.g. while number &gt; 1:     number -= 1     print("The number is now", number) </pre>
<p><u>Post-condition iteration (repeat-until)</u></p> <p><i>This is also to repeatedly do tasks, while some condition is true, however the condition is checked after the code is run and not before.</i></p> <p><i>In pseudocode, these post-condition loops have an inverted condition, meaning that it does something <b>until</b> the condition is true, <b>not while</b> it is true.</i></p>	<pre> REPEAT     &lt;code&gt; UNTIL &lt;condition&gt;  // e.g. REPEAT     OUTPUT "Enter the password..."     INPUT Password     IF Password &lt;&gt; "Secret"         THEN             OUTPUT "Wrong..."         ENDIF UNTIL Password = "Secret" </pre>	<pre> # Repeat-until loops do not exist in # Python due to it being mostly # redundant. You cannot do post- # condition loops either. You can # replicate the example like so:  # negate the condition while password != "Secret":     password = input("Enter the password...")     if password != "Secret":         print("Wrong...") </pre>



## Arrays

*This is used to store sequences of data, or grids/matrices of data.*

**Arrays in Pseudocode begin at 1, and they begin at 0 in Python.**

```
// In Pseudocode, arrays are STATIC,
// meaning that you cannot add or
// remove elements dynamically.
//
// Declaring an ARRAY (1 dimensional)
//
// l is the lower bound, h is the
// higher bound
DECLARE <ident>: ARRAY[l,h] OF <type>

// Declaring an ARRAY (2 dimensional)
//
// l1 and h1 are the bounds of the
// first dimension, l2 and h2 are the
// bounds of the second dimension
DECLARE <ident>: ARRAY[l1,h1:l2,h2] OF
<type>

// e.g.
DECLARE StudentNames: ARRAY[1,5] OF
STRING

// Adapted from the IGCSE Syllabus
DECLARE TicTacToe: ARRAY[1,3:1,3] OF
CHAR

// Assign to an ARRAY (1 dimensional)
StudentNames[2] ← "Marcos"
TicTacToe[1,3] ← 'X'

// Use an ARRAY
<ident>[<index>] // 1D ARRAY
<ident>[<index1>,<index2>] // 2D ARRAY

// e.g.
StudentNames[3] // get 3rd student name
TicTacToe[2,1] // get the character at
                // 2, 1 on the Tic Tac
                // Toe board
```

```
# Python does not have pseudocode
# ARRAYS, i.e. sequences of data of a
# fixed length, however, Python does
# have lists with push-back/pop-back
# functionality.
#
# You must also initialize every list
# before using them!
#
# Declaring a list (1 dimensional)

# you do not have to specify bounds!
<ident>: list[<type>]

# Declaring a list (2 dimensional)
<ident>: list[list[<type>]]

# Initializing a list (1D):
<ident> = []

# Initializing a list (2D)
<ident> = [[]]

# e.g.
student_names: list[str]

# Python does not have CHAR!
tic_tac_toe: list[list[str]]

# Assign to a list
student_names[2] = "Marcos"

# You can even assign a whole list!
student_names = ["Tom", "James",
"Jimmy", "John", "Peter"]

# Use a list
<ident>[<index>] # 1D list
<ident>[<index1>][<index2>] # 2D list

# e.g.
student_names[3] # get 3rd student
                 # name
tic_tac_toe[2][1] # get the character
                 # at 2, 1 on the
                 # Tic Tac Toe board
```

<p><u>Iteration (for)</u>  <i>This is to repeatedly do something until a counter reaches the end, which is specified.</i></p>	<pre> FOR &lt;counter&gt; ← &lt;begin&gt; TO &lt;end&gt;     &lt;code&gt; NEXT &lt;counter&gt;  FOR &lt;counter&gt; ← &lt;begin&gt; TO &lt;end&gt; STEP &lt;step&gt;     &lt;code&gt; NEXT &lt;counter&gt;  // e.g.  // Assume LENGTH() calculates the // length of an array FOR Counter ← 1 TO LENGTH(StudentNames)     OUTPUT "There is a student called", StudentNames[Counter], " in the class." NEXT Counter  FOR OddNumber ← 1 TO 30 STEP 2     OUTPUT OddNumber NEXT OddNumber </pre>	<pre> for &lt;counter&gt; in range(&lt;begin&gt;, &lt;end&gt;):     &lt;code&gt;  for &lt;counter&gt; in range(&lt;begin&gt;, &lt;end&gt;, &lt;step&gt;):     &lt;code&gt;  # lists in Python begin at 0! # e.g. for counter in range(0, len(student_ names)):     print("There is a student called ", student_names[counter], "in the class.")  for odd_number in range(1, 30, 2):     print(odd_number) </pre>
<p><u>Procedures</u>  <i>These are repeatable sections of code that can be invoked (called) over and over as many times as needed. This might also be called a subprogram, or a subroutine (outdated).</i></p>	<pre> // declaring procedures PROCEDURE &lt;name&gt;     &lt;code&gt; ENDPROCEDURE  PROCEDURE &lt;name&gt;(&lt;parameter name&gt;: &lt;type&gt;, &lt;parameter name&gt;:&lt;type&gt;, ...)     &lt;code&gt; ENDPROCEDURE  // e.g. PROCEDURE SayHello     OUTPUT "Hello!" ENDPROCEDURE  PROCEDURE Line(Size:INTEGER)     FOR Length ← 1 TO Size         OUTPUT '-'     NEXT Length ENDPROCEDURE  // calling procedures CALL &lt;name&gt; CALL &lt;name&gt;(&lt;parameter&gt;, &lt;parameter&gt;...)  // e.g. CALL SayHello CALL Line(10) </pre>	<pre> # all "procedures" below are # technically functions, as Python # does not differentiate between # Procedures and Functions.  # declaring procedures def &lt;name&gt;():     &lt;code&gt;  def &lt;name&gt;(&lt;parameter name&gt;:&lt;type&gt;, &lt;parameter name&gt;:&lt;type&gt;, ...):     &lt;code&gt;  # e.g. def say_hello():     print("Hello!")  def line(size: int):     for length in range(1, size):         print('-')  # calling functions &lt;name&gt;() &lt;name&gt;(&lt;parameter&gt;, &lt;parameter&gt;...)  # e.g. say_hello() line(10) </pre>

<p><u>Functions</u>  <i>These are repeatable sections of code, but they <b>return</b> values, meaning that they usually <b>process</b> or <b>give</b> data back to the site of invocation, also known as the caller.</i></p> <p><i>Procedures can also be referred to as <b>fruitless</b> and Functions <b>fruitful</b> due to functions requiring a return value.</i></p> <p><i>Python does not differentiate between functions and procedures.</i></p>	<pre>// declaring functions <b>FUNCTION</b> &lt;name&gt; <b>RETURNS</b> &lt;type&gt;     &lt;code&gt;     <b>RETURN</b> &lt;expr&gt; // you MUST return                     // something!  <b>ENDFUNCTION</b>  <b>FUNCTION</b> &lt;name&gt;(&lt;parameter name&gt;: &lt;type&gt;, &lt;parameter name&gt;:&lt;type&gt;, ...) <b>RETURNS</b> &lt;type&gt;     &lt;code&gt;     <b>RETURN</b> &lt;expr&gt; // you MUST return                     // something!  <b>ENDFUNCTION</b>  // e.g. <b>FUNCTION</b> GimmeFive <b>RETURNS</b> <b>INTEGER</b>     <b>RETURN</b> 5 <b>ENDFUNCTION</b>  <b>FUNCTION</b> AddOne(Num:<b>INTEGER</b>) <b>RETURNS</b> <b>INTEGER</b>     <b>DECLARE</b> Result:<b>INTEGER</b>     Result ← Num + 1     <b>RETURN</b> Result <b>ENDFUNCTION</b>  // calling functions GimmeFive() AddOne(5)  // ...or use them as expressions AddOne(GimmeFive()) <b>OUTPUT</b> GimmeFive(), "+ 1 is", AddOne(5)</pre>	<pre># declaring functions <b>def</b> &lt;name&gt;() -&gt; &lt;type&gt;:     &lt;code&gt;     <b>return</b> &lt;expr&gt; # you MUST return                   # something!  <b>def</b> &lt;name&gt;(&lt;parameter name&gt;:&lt;type&gt;, &lt;parameter name&gt;:&lt;type&gt;, ...) -&gt; &lt;type&gt;:     &lt;code&gt;     <b>return</b> &lt;expr&gt; # you MUST return                   # something!  # e.g. <b>def</b> gimme_five() -&gt; <b>int</b>:     <b>return</b> 5  <b>def</b> add_one(num: <b>int</b>) -&gt; <b>int</b>:     result: <b>int</b>     result = num + 1     <b>return</b> result  # calling functions gimme_five() add_one(5)  # ...or use them as expressions add_one(gimme_five()) <b>print</b>(gimme_five(), "+ 1 is", add_one(5))</pre>
<p><u>File I/O</u>  <i>Self explanatory. This relates to writing data and reading data from files on the disk, hard drive, etc. that is <b>not in memory</b>.</i></p>	<pre>// file modes include READ and WRITE // // opening files <b>OPENFILE</b> &lt;file name&gt; <b>FOR</b> &lt;file mode&gt;  // reading files (read into &lt;variable&gt;) <b>READFILE</b> &lt;file name&gt;, &lt;variable&gt;  // writing files (write from &lt;variable&gt;) <b>WRITEFILE</b> &lt;file name&gt;, &lt;variable&gt;  // closing files <b>CLOSEFILE</b> &lt;file name&gt;  // e.g. <b>OPENFILE</b> data.txt <b>FOR</b> <b>READ AND WRITE</b> <b>READFILE</b> data.txt, Content <b>WRITEFILE</b> data.txt, Content + "Hi!" <b>CLOSEFILE</b> data.txt</pre>	<pre># READ corresponds to 'r' # WRITE corresponds to 'w' # READ AND WRITE corresponds to 'r+' # or 'w+' # opening files &lt;ident&gt; = <b>open</b>(&lt;file name&gt;, &lt;file mode&gt;)  # reading files &lt;variable&gt; = &lt;ident&gt;.<b>read</b>()  # writing files &lt;ident&gt;.<b>write</b>(&lt;variable&gt;)  # closing files &lt;ident&gt;.<b>close</b>()  # e.g. file = <b>open</b>("data.txt", "r+") content = file.<b>read</b>() file.<b>write</b>(content + "Hi!") file.<b>close</b>()</pre>

# Appendix

The QR code for the online copy is found below.

It is hosted on my website, [ezntek.com](https://ezntek.com).



Alternatively, find it [here](#).

(The URL is [https://ezntek.com/revision/pseudocode\\_reference.html](https://ezntek.com/revision/pseudocode_reference.html))

The blog post, which has some more information, may be found [here](#).

(The URL is <https://ezntek.com/posts/the-igcse-pseudocode-to-python-reference-guide-for-g1-and-g2-computer-science-20241018t2049/>)