The Pocket IGCSE Pseudocode to Python Reference Guide

Eason Qin Luojia (eason@ezntek.com)

Fourth Revision

October 22nd, 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 **variable name or value, function calls, or arithmetic expressions, enclosed or not enclosed in brackets**. It **will** be shortened to **expr** when necessary.
Identifier `<ident>`	A **variable name**. It **will** be shortened to **ident** 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

Reference Guide

Item	IGCSE Pseudocode	Python

Comment
Used to annotate code.
	`// This is a comment.`
	`// To comment,  simply put two`
	`// slashes (//) in front of your text.`
		`# This is a comment.`
		`# To comment, simply put one`
		`# hashtag (#) in front of your`
		`# text.`

Values
Also known as Literals, they represent values.
	`// These are all INTEGER's, or whole`
	`// numbers`
	`42`
	`-2043`
	
	`// These are all REAL's, or decimal`
	`// numbers`
	`3.14159`
	`2.718282`
	`56.52`
	
	`// These are STRING's, or "text"`
	`// (enclosed in only "):`
	`"Good morning, user!"`
	`"Thomas"`
	`"Jason Lee"`
	
	`// These are BOOLEAN's, either TRUE or FALSE`
	`TRUE`
	`FALSE`
	
	`// These are CHAR's, or singular`
	`// characters (enclosed only in '):`
	`'c'`
	`'F'`
	`'b'`
		`# These are all int's, or whole`
		`# numbers`
		`42`
		`-2043`
		
		`# these are all float's, or decimal`
		`# numbers`
		`3.14159`
		`2.718282`
		`56.52`
		
		`# These are str's, or "text"`
		`# (enclosed in both " and ')`
		`"Good morning, user!"`
		`'Thomas'`
		`'Jason Lee'`
		
		`# These are bool's, either TRUE or`
		`# FALSE`
		`True`
		`False`
		
		`# there is no CHAR in Python, just use a str.`

Declaring a variable
This is to make it clear to the computer that the variable exists.
 	
This is not necessary in Python.
	`DECLARE <variable name>: <type>`
	
	`// e.g.`
	`DECLARE Name: STRING`
	`DECLARE TotalScore: INTEGER`
	`// or,`
	`DECLARE Name:STRING`
	`DECLARE TotalScore:INTEGER`
		`<variable name>: <type>`
		
		`# e.g.`
		`name: str`
		`total_score: int`

Assignment
This is used to give a value to a previously declared variable.
	<variable name> ← <expression>
	// NOTE: you may write it like <- in
	// your computer.
	
	// e.g.
	Name ← "Thomas"
	TotalScore ← 84
	Name ← FirstName
		<variable name> = <expression>
		
		# e.g.
		name = "Thomas"
		total_score = 84
		name = first_name

Input and Output
This is used to give users feedback and receive input.
	OUTPUT <expression>
	OUTPUT <expression>, ...
	// Print however many things you
	// require.
	
	INPUT <expression>
	
	// e.g.
	OUTPUT "What is your name"
	OUTPUT "Welcome", Name
	OUTPUT "What is your Social Security Number?"
	INPUT SocialSecurityNumber
	OUTPUT "What is your ID?"
	INPUT ID
		print(<expression>)
		print(<expression>, ...)
		# Print however many things you
		# require.
		
		<variable name> = input(<prompt>)
		
		# e.g.
		print("What is your name")
		print("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 = int(input())
		id = input("What is your ID?")
		id = int(id)

Arithmetic (expression)
This is to do math.
	<expr> <operator> <expr>
	
	// e.g.
	2 + 5
	(3 * X) + 1
	
	// you can combine it with an
	// asasignment, like so:
	NextTerm ← X + 1
		<expr> <operator> <expr>
		
		# e.g.
		2 + 5
		(3 * x) + 1
		
		# you can combine it with an
		# assignment, like so:
		next_term = x + 1

Arithmetic Assignments
This is to perform a math operation on the variable itself, including incrementing a variable, etc.
	// They DO NOT exist in pseudocode,
	// but may be substituted with:
	
	<ident> ← <ident> <operator> <expr>
	
	// e.g.
	Age ← Age + 1
	Temperature ← Temperature - 5
		<ident> <operator>= <expr>
		
		# e.g.
		age += 1
		temperature -= 5

Comparison Operators
This is to check the relation between two values, such as equality, greater or less than, not equal to, etc.
	// Equality
	Age = 18
	
	// Greater than, less than
	Age > 18
	Age < 18
	
	// Greater than or equal to, less
	// than or equal to
	Age >= 18
	Age <= 18
	
	// Not equal to
	Age <> 18
		# Equality
		age == 18
		
		# Greater than, less than
		age > 18
		age < 18
		
		# Greater than or equal to, less
		# than or equal to
		age >= 18
		age <= 18
		
		# Not equal to
		age != 18

Boolean Expressions
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.
	// 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
		# 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

Conditional Branching (if)
This is to make a decision, a choice, to ask a question, whichever interpretation pleases you.
	// either:
	IF <condition>
	  THEN // PRESS SPACE TWICE!
	  <code> // PRESS SPACE TWICE!
	ELSE
	  <code> // PRESS SPACE TWICE!
	ENDIF
	
	// or:
	IF <condition>
	  THEN
	  <code>
	ENDIF
	
	// e.g.
	IF Age > 18
	  THEN
	  OUTPUT "you can drink!"
	ELSE
	  OUTPUT "you cannot drink..."
	ENDIF
		if <condition>:
		  <code> # PRESS SPACE 4 TIMES!
		else:
		  <code>
		
		# or
		if <condition>:
		  <code>
		
		# e.g.
		if age > 18:
		  print("you can drink!")
		else:
		  print("you cannot drink...")

<table>
<tr>
<td><u>Chained conditional branching (if-else if-else)</u>
This is to ask multiple questions in a row.

Note that in pseudocode, you must follow this indentation exactly, i.e. THEN must be on a new line and indented by 2 spaces, and the code block by 4, ELSE by none, and the code block that follows by 2.

ALL OTHER CODE BLOCKS ARE INDENTED BY 4 SPACES.</td>
<td><pre>
// This does not exist in pseudocode, but can be emulated in the following way:

IF <condition>
  THEN
    <code>
ELSE
  IF <condition>
    THEN
      <code>
  ELSE
    <code>
ENDIF

// with the IF statement inside the
// larger ELSE statement being able
// to be repeated as many times as
// needed.

IF Age > 18
  THEN
    OUTPUT "You can drink!"
ELSE
  IF Age > 16
    THEN
      OUTPUT "You can almost drink!"
  ELSE
    OUTPUT "You can't drink..."
ENDIF
</pre></td>
<td><pre>
if <condition>:
    <code>
elif <condition>:
    <code>
else:
    <code>

# e.g.
if age > 18:
    print("you can drink!")
elif age > 16:
    print("you can almost drink!")
else:
    print("you can't drink...")
</pre></td>
</tr>
<tr>
<td><u>Pattern Matching</u>
This is like finding a value that matches the one that you have, and then doing something when you find it.
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.</td>
<td><pre>
CASE OF <expr>
  <expr>: <statement>
  <expr>: <statement>
  ...
  // optionally,
  OTHERWISE <statement>
ENDCASE

// e.g.
CASE OF BottleMaterial
  "Plastic": OUTPUT "Unsustainable..."
  "Metal": OUTPUT "Sustainable!"
  "Glass": OUTPUT "Fragile..."
  "Paper": OUTPUT "WHY?"
  OTHERWISE OUTPUT "Unrecognized"
ENDCASE
</pre></td>
<td><pre>
match <expr>:
    case <expr>:
        <code>
    case <expr>:
        <code>
    ...
    # This is equivalent to OTHERWISE
    case _:
        <code>

match bottle_material:
    case "Plastic":
        print("Unsustainable...")
    case "Metal":
        print("Sustainable!")
    case "Glass":
        print("Fragile...")
    case "Paper":
        print("WHY?")
    case _:
        print("Unrecognized")
</pre></td>
</tr>
</table>

Pre-condition iteration (while)
 	
This is to repeatedly do tasks, while some condition is true (so to not infinitely loop).
	``
	WHILE <condition> DO
	    <code>
	ENDWHILE
	
	// e.g.
	WHILE Number > 1 DO
	    Number ← Number - 1
	    OUTPUT "The number is now", Number
	ENDWHILE
	``
		``
		while <condition>:
		    <code>
		
		# e.g.
		while number > 1:
		    number -= 1
		    print("The number is now",number)
		``

Post-condition iteration (repeat-until)
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.
 	
*In pseudocode, these post-condition loops have an inverted condition, meaning that it does something **until** the condition is true, **not while** it is true.*
	``
	REPEAT
	    <code>
	UNTIL <condition>
	
	// e.g.
	REPEAT
	    OUTPUT "Enter the password..."
	    INPUT Password
	    IF Password <> "Secret"
	      THEN
	        OUTPUT "Wrong..."
	    ENDIF
	UNTIL Password = "Secret"
	``
		``
		# 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...")
		``

Arrays

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

```
// 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
```

Iteration (for)
This is to repeatedly do something until a counter reaches the end, which is specified.
	``
	FOR <counter> ← <begin> TO <end>
	    <code>
	NEXT <counter>
	
	FOR <counter> ← <begin> TO <end> STEP <step>
	    <code>
	NEXT <counter>
	
	// 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
	``
		``
		for <counter> in range(<begin>, <end>):
		    <code>
		
		for <counter> in range(<begin>, <end>, <step>):
		     <code>
		
		# e.g.
		for counter in range(1, 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)
		``

Procedures
*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).*
	``
	// declaring procedures
	PROCEDURE <name>
	    <code>
	ENDPROCEDURE
	
	PROCEDURE <name>(<parameter name>:<type>, <parameter name>:<type>, ...)
	    <code>
	ENDPROCEDURE
	
	// e.g.
	PROCEDURE SayHello
	    OUTPUT "Hello!"
	ENDPROCEDURE
	
	PROCEDURE Line(Size:INTEGER)
	    FOR Length ← 1 TO Size
	        OUTPUT '-'
	    NEXT Length
	ENDPROCEDURE
	
	// calling procedures
	CALL <name>
	CALL <name>(<parameter>, <parameter>...)
	
	// e.g.
	CALL SayHello
	CALL Line(10)
	``
		``
		# all "procedures" below are
		# technically functions, as Python
		# does not differentiate between
		# Procedures and Functions.
		
		# declaring procedures
		def <name>():
		    <code>
		
		def <name>(<parameter name>:<type>, <parameter name>:<type>, ...):
		    <code>
		
		# e.g.
		def say_hello():
		    print("Hello!")
		
		def line(size: int):
		    for length in range(1, size):
		        print('-')
		
		# calling functions
		<name>()
		<name>(<parameter>, <parameter>...)
		
		# e.g.
		say_hello()
		line(10)
		``

Functions

*These are repeatable sections of code, but they **return** values, meaning that they usually **process** or **give** data back to the site of invocation, also known as the caller.*

*Procedures can also be referred to as **fruitless** and Functions **fruitful** due to functions requiring a return value.*

Python does not differentiate between functions and procedures.

```
// declaring functions
FUNCTION <name> RETURNS <type>
    <code>
    RETURN <expr> // you MUST return
                  // something!
ENDFUNCTION

FUNCTION <name>(<parameter name>:
<type>, <parameter name>:<type>, ...)
RETURNS <type>
    <code>
    RETURN <expr> // you MUST return
                  // something!
ENDFUNCTION

// e.g.
FUNCTION GimmeFive RETURNS INTEGER
    RETURN 5
ENDFUNCTION

FUNCTION AddOne(Num:INTEGER) RETURNS
INTEGER
    DECLARE Result:INTEGER
    Result ← Num + 1
    RETURN Result
ENDFUNCTION

// calling functions
GimmeFive()
AddOne(5)

// ...or use them as expressions
AddOne(GimmeFive())
OUTPUT GimmeFive(), "+ 1 is", AddOne(5)
```

```
# declaring functions
def <name>() -> <type>:
    <code>
    return <expr> # you MUST return
                  # something!

def <name>(<parameter name>:<type>,
<parameter name>:<type>, ...) ->
<type>:
    <code>
    return <expr> # you MUST return
                  # something!


# e.g.
def gimme_five() -> int:
    return 5

def add_one(num: int) -> int:
    result: int
    result = num + 1
    return result

# calling functions
gimme_five()
add_one(5)

# ...or use them as expressions
add_one(gimme_five())
print(gimme_five(), "+ 1 is",
add_one(5))
```

File I/O

*Self explanatory. This relates to writing data and reading data from files on the disk, hard drive, etc. that is **not in memory**.*

```
// file modes include READ and WRITE
//
// opening files
OPENFILE <file name> FOR <file mode>

// reading files (read into <variable>)
READFILE <file name>, <variable>

// writing files (write from
<variable>)
WRITEFILE <file name>, <variable>

// closing files
CLOSEFILE <file name>

// e.g.
OPENFILE data.txt FOR READ AND WRITE
READFILE data.txt, Content
WRITEFILE data.txt, Content + "Hi!"
CLOSEFILE data.txt
```

```
# READ corresponds to 'r'
# WRITE corresponds to 'w'
# READ AND WRITE corresponds to 'r+'
# or 'w+'
# opening files
<ident> = open(<file name>, <file
mode>)

# reading files
<variable> = <ident>.read()

# writing files
<ident>.write(<variable>)

# closing files
<ident>.close()

# e.g.
file = open("data.txt", "r+")
content = file.read()
file.write(content + "Hi!")
file.close()
```

Appendix

The QR code for the online copy is found below.

It is hosted on my website, ezntek.com.

Alternatively, find it here.

(The URL is 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/)