PythoniC Documentation

Release 0.0.1

lightsing, Cubic Pills

Apr 16, 2017

Contents:

PythoniC is a variant of *C language*. With writing code in *Python* style, you can enjoy your life with *C*‘s efficiency.

You should realize that *PythoniC* is still compiled by c compiler. So, *PythoniC* should has the same behaviour as *C* rather than *Python*. For example, you shoud care about the type of your variables, the range of *int* numbers, etc. Alought *Python* style can bring you convenient , you should realize this is based on setting restrictions on your code. You should not write hacking code anymore. “Please think problems in *Python*.”

CHAPTER 1

Reference Guide

PythoniC Keywords

PythoniC Part

Keyword	Description	Example
and	Logical and.	`True and False == False`
break	Stop this loop right now.	`while True:  break`
bool	Boolean type	
class	Define a class.	`class Person(object)`
continue	Don't process more of the loop, do it again.	`while True:  continue`
def	Define a function.	`def X():  pass`
elif	Else if condition.	`if:  X; elif:  Y; else:  J`
else	Else condition.	`if:  X; elif:  Y; else:  J`
for	Loop over a collection of things.	`for X in Y: pass`
from	Include header from subdirectory.	`from foo import bar`
is	Same as ==	`a = 1; a is 1 == True`
if	If condition.	`if:  X; elif:  Y; else:  J`
import	Include header	`import stdio`
in	Part of for-loops. Also a test of X in Y.	`for X in Y: pass` also `1 in [1] == True`
lambda	Create a short anonymous function.	`s = lambda y:  y ** y; s(3)`
not	Logical not.	`not True == False`
or	Logical or.	`True or False == True`
pass	Do nothing	`def empty():  pass`
print	Print this string.	`print("Hello")`
return	Exit the function with a return value.	`def X():  return Y`
while	While loop.	`while X: pass`

C Part

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	sizeof	volatile	goto
do	if	static	while

`goto` is no longer supported in *PythoniC*, but it still is a part of *PythoniC* keywords.

PythoniC Identifiers

PythoniC Identifiers are the same as *C* Identifiers.

“Identifiers” or “symbols” are the names you supply for variables, types, functions, and labels in your program. Identifier names must differ in spelling and case from any keywords. You cannot use keywords as identifiers; they are reserved for special use. You create an identifier by specifying it in the declaration of a variable, type, or function. In this example, *result* is an identifier for an integer variable, and *main* and *print* are identifier names for functions.

C code

```
#include <stdio.h>

int main() {
    int result;
    if(result != 0) {
        printf( "Bad file handle\n" );
    }
    return 0;
}
```

PythoniC Code

```
import stdio

def main() -> int:
    int result
    if result != 0:
        print( "Bad file handle" )
    return 0
```

Once declared, you can use the identifier in later program statements to refer to the associated value.

PythoniC Lexical Analysis

PythoniC learn from Python’s design ideas, as much as possible to implement similar designs with Python on the premise of guaranteeing usability.

Encoding

Considering the compatibility, donnot put non-ASCII characters in your source code (except for comments). **The default encoding is UTF-8.**

Comment for encding declarations isn't supported. If the file cannot be decoded, an `Encoding Error` will be throw out.

Line structure

A PythoniC program is divided into a number of logical lines.

Logical lines

The end of a logical line is represented by the token NEWLINE. Statements cannot cross logical line boundaries except where NEWLINE is allowed by the syntax (e.g., between statements in compound statements). A logical line is constructed from one or more *physical lines* by following the explicit or implicit *line joining* rules.

Physical lines

A physical line is a sequence of characters terminated by an end-of-line sequence. In source files, any of the standard platform line termination sequences can be used - the Unix form using ASCII LF (linefeed), the Windows form using the ASCII sequence CR LF (return followed by linefeed), or the old Macintosh form using the ASCII CR (return) character. All of these forms can be used equally, regardless of platform.

Comments

A comment starts with a hash character (`#`) that is not part of a string literal, and ends at the end of the physical line. A comment signifies the end of the logical line unless the implicit line joining rules are invoked. Comments are ignored by the syntax; they are not tokens.

Specially, C style comment is not supported in PythoniC. Including single line comment (`//`) and multiline comment (`/* */`).

Explicit line joining

Two or more physical lines may be joined into logical lines using backslash characters (`\`), as follows: when a physical line ends in a backslash that is not part of a string literal or comment, it is joined with the following forming a single logical line, deleting the backslash and the following end-of-line character. For example:

```
if 1900 < year < 2100 and 1 <= month <= 12 \
  and 1 <= day <= 31 and 0 <= hour < 24 \
  and 0 <= minute < 60 and 0 <= second < 60:   # Looks like a valid date
    return 1
```

A line ending in a backslash cannot carry a comment. A backslash does not continue a comment. A backslash does not continue a token except for string literals (i.e., tokens other than string literals cannot be split across physical lines using a backslash). A backslash is illegal elsewhere on a line outside a string literal.

Blank lines

A logical line that contains only spaces, tabs, formfeeds and possibly a comment, is ignored (i.e., no NEWLINE token is generated). During interactive input of statements, handling of a blank line may differ depending on the implementation of the read-eval-print loop. In the standard interactive interpreter, an entirely blank logical line (i.e. one containing not even whitespace or a comment) terminates a multi-line statement.

Indentation

Same as Python, PythoniC won't use curly brackets (`{}`).

Leading whitespace (spaces and tabs) at the beginning of a logical line is used to compute the indentation level of the line, which in turn is used to determine the grouping of statements. The recommend indentation for one group is 4 spaces.

Tabs are replaced (from left to right) by one to eight spaces such that the total number of characters up to and including the replacement is a multiple of eight (this is intended to be the same rule as used by Unix). The total number of spaces preceding the first non-blank character then determines the line's indentation. Indentation cannot be split over multiple physical lines using backslashes; the whitespace up to the first backslash determines the indentation.

Indentation is rejected as inconsistent if a source file mixes tabs and spaces in a way that makes the meaning dependent on the worth of a tab in spaces; a `Tab Error` will be throw out. in that case.

Cross-platform compatibility note: because of the nature of text editors on non-UNIX platforms, it is unwise to use a mixture of spaces and tabs for the indentation in a single source file. It should also be noted that different platforms may explicitly limit the maximum indentation level.

The indentation levels of consecutive lines are used to generate INDENT and DEDENT tokens, using a stack, as follows.