Quick Guide to Python for C programmer

For full documentation, you should always reference https://www.python.org/doc/

Do note: Version 3.x is NOT backward compatible with Version 2.x

Last update : February, 2017

CONTENTS

SCOPE 3

SET UP: 4

IDE for Pyton 4

Some Quick LInks 5

MOST BASIC INTRINSIC TYPES IN PYTHON 6

List of most basic differences between C/C++ and Python 6

Simple Standard Input 10

SOME STANDARD DATA TYPES 11

Sequence Types 12

with string with " " 12

with List with [ ] 13

with Tuples with ( ) 14

Bytearray 14

Mapping Types 16

with dictionary with { 'key1' : 'value', 'key2' : i-value, etc } 16

Set Types 17

with sets with set (immutable objects) 17

About Data type conversion 18

SCOPE

1. This document assumes you have already had a sound foundation in C/C++.
2. This document guides you a quick transition into using Python, not meant to be a full blown tutorial.
3. Far more importantly, you should learn how to be resourceful to look up the latest update.
4. Do note that version 3 is not compatible with version 2. Therefore, you will need to always refer back to the official Python Documentation - https://docs.python.org .

SET UP:

1) For windows system, edit setvar.bat:

```
set PYTHONPATH=c:\Python27
set PYTHONPATH=%PYTHONPATH%;C:\PYTHONPATH\libs;c:\PYTHONPATH\Lib
rem DLL  %PYTHONPATH%\DLLs

set path=%path%;%PYTHONPATH%\bin
```

2) Configuration files that you should be aware of. You do not need to customize it. See www.python.org for details if you want to customize it.

   py.ini, pyvenv.cfg, PYTHONHOME, PYTHONPATH

3) Change python code to executable ?

   Cx_Freeze is a distutils extension (see Extending Distutils) which wraps Python scripts into executable.

4) Byte-compile Python libraries

   — Compile a single file : *python -m* (Lib/py_compile.py) - to generate a byte-code file from a source file

   — Compile multiple files : *python -m compileall* (Lib/compileall.py)

I'll add more into individual modules installation later. For more detailed information, you may refer to :

https://docs.python.org/2/installing/index.html

IDE FOR PYTON

- Visual Studio 2015 and up
- Eclipse plugin:
  - Install Eclipse Neon
  - go to http://marketplace.eclipse.org/marketplace-client-intro?mpc_install=114 for installing the plug-in.
- GEdit – a platform just for python

Official Python Documentation:

https://docs.python.org/2/index.html

Basic describes syntax and language Reference:

https://docs.python.org/2/reference/index.html

Standard Library functions:

https://docs.python.org/2/library/index.html

About Constants :

https://docs.python.org/2/library/constants.html

Built-in Functions:

https://docs.python.org/2/library/functions.html

MOST BASIC INTRINSIC TYPES IN PYTHON

You should always refer to the official document for the proper version to see the full list, and the latest update. https://docs.python.orge.

LIST OF MOST BASIC DIFFERENCES BETWEEN C/C++ AND PYTHON

Here lists major differences in most commonly used language structure between Python and C/C++.

Do note:

C/C++	Python
In Windows, you need the installation folder in your system environment variable $PATH	In linux/unix, you need this at top of a file: #!/usr/bin/python
*.c or *.cpp	*.py
Explicit Pointers reference	Almost everything in Python is in "object" reference. No explicit pointer reference.
int x = 10;	x = 10
single quote is different from double quote char *x = 'hello'; NOOOO! char *x = "hello";	single quote is treated the same as double quote e.g. x = 'hello' is the same as x= "hello"
void main()…	Do not need it. Expressions have no preceding space are considered to be in "main" per se
{… }	Indentation only All statements within a block must be indented with the same amount of space.
while (I < 10) { …. }	while (I < 10) : // a scope is identified by indentation.
#include <stdio.h>	Import sys
// comment /* block of comment */	# comment ''' block of comment '''
primitive data type: int, float, long, char, etc.	There is really no such thing as primitive data types. The most fundamental types are in objects context. So, they are like pointer-reference.

	These are some Python's standard data types: Numeric types (they are treated like reference too): int, float, long (e.g. 5199L), complex (e.g. 4.5j) Sequences: String, byte array (3), list, tuple Sets and mappings: Such as List, Dictionary
#include	From *module* import * e.g. if you want a c/c++ type of array implementation, you need to include the following : from array import *
You cannot delete a variable. However, you can get the same effect using scope rule, i.e. {…. } { int varX = 1; }	varX = 1 del varX
x = x + 10 + 20 + 30;	x = x + 10 \ + 20 \ + 30;
Functions… e.g. int factorial(num) {… } void displayResult(char *arr) { … }	No function return type. Just use "def" e.g. def factorial(num): you code here must be indented return result def displayResult(arr) the function code return
char days[][10] = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday"};	days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
char sentence[] = "It is a sentence";	sentence = "It is a sentence."
int counter = 10; printf("%d\n", x);	Counter = 10 Print counter + '\n'
printf("\nPress Enter to exit."); while (getchar() != '\n') ;	raw_input("\nPress Enter to exit.")

	This raw input can be changed into the data type needed as well. See the sample after this table.
char x[] = “foo”; printf(“%s\n”, x);	x = 'foo'; sys.stdout.write(x + '\n')
printf(“%s age is %d”, name, num);	print “%d age is %d”, name, num,

char x[ ] = “abcde”;
while (*x!=0)
{ printf(“%c,”, *x);
x++;
}
	mylist = “abcde”;
	
	for p in mylist:
	print p,
	or
	print ",".join([str(p) for p in mylist])
	
	do note:
	mylist2 = "abcdefg", output will be *a, b, c, d, e, f, g*
	but
	mylist2 = [*"abcdefg"*], output will be abcdefg

else if	elif:
int a = 1, b = 2; char *c = “John”;	a, b, c = 1, 2, "John"
&&	and
||	or
! (	not

Operators in Python do not exist in C or C++

Power: ** e.g. 5**2 =25 B = B**2 B **=2
Floor Division: // e.g. 10 // 3 = 3 X = X//3 X //=3
Membership:

is is not	Identity operators
in not in	Membership operators

not or and	Logical operators

e.g.

```
a = 10
b = 20
mylist = [1, 2, 3, 4, 5 ]

if  a in list :
    print a + ' is in mylist'

if  a not in list:
    print a, " is not in mylist"
```

SIMPLE STANDARD INPUT

```
def userInput():
    name = input("Your Name? ")
    fruitlist = raw_input("Enter [ list of your favourite fruit ] : ")
    print name, "'s favourite fruit are: ",  fruitlist
    return
```

Console: Green is your input. Note that input data has to conform to the python data type syntax.

```
Your Name? "Elizabeth"
Enter [ list of your favourite fruit ] : [orange, apple, kiwi, watermelon]
Elizabeth 's favourite fruit are:  [orange, apple, kiwi, watermelon]
```

REFERENCE:

More about raw_input : https://docs.python.org/2/library/functions.html#raw_input

See the full list of Built-in Functions :
https://docs.python.org/2/library/functions.html#built-in-functions

SOME STANDARD DATA TYPES

Python has five standard data types –

- Numbers
- String
- List
- Tuple
- Dictionary

View all intrinsic data types for version 2.7.*:
https://docs.python.org/2/library/datatypes.html

View all intrinsic standard data types for version 3.6+:
https://docs.python.org/3/library/datatypes.html

Installing Modules

In Windows

To use Python under Windows, you need to install the Windows binary installer, which you can download from the Python download page. Make sure you choose the binary installer.

SEQUENCE TYPES

As far as up to version 2.7, there are seven sequence types. You should always refer to the official document for the proper version to see the full list, and the latest update.

e.g. https://docs.python.org/2/library/stdtypes.html#sequence-types-str-unicode-list-tuple-bytearray-buffer-xrange.

The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator.

Basic mechanic : (https://docs.python.org/2/tutorial/introduction.html)

- Indexing, 0 means 1st element. Can use negative, that means from the end which starts with -1.
- Slicing, such as getting substring, sub-elements, etc.
- Matrixes, ie. array operations.

Only the following will be discussed in this document: str, list, tuple and bytearray.

WITH STRING *WITH " "*

```
str = 'Hello World!'
print str          # Hello World!
print str[0]       # H
print str[2:5]     # llo
print str[2:]      # llo World!
print str[-1]      # '!'
print str[-6:-2]   # 'World'
print str[:-8]     # 'Hello'
print str * 2      # Hello World!Hello World!
print str + "!!"   # Hello World!!!
```

WITH LIST *WITH [ ]*

A list contains an ordered collection of mutable objects.

A single object is almost like a single node of C's struct with various data types.

Delete and Insert operations are expensive, as the list re-orders itself.

See online document for all Built-in List Functions and Methods.

```
#!/usr/bin/python

list_long = [ 'robot', 100 , 3.1415, 'John', 1.61803 ]
list_short = [123, 'John']
list_short[1] = "Moe"

print list + tinylist          #    ['robot', 100, 3.1415, 'John', 1.6180300000000000003, 123,
'Moe']
del  list_long[2]

print list_long                #    ['robot', 100, 'John', 1.6180300000000000003, 123, 'Moe']
print len( list_long )         #   4
print 3 in list_long           #   1.61803
for x in list_long :
    print x                    #       'robot 100  3.1415  1.61803
list_short.insert(1, 'and')
print list_short               #  [123, 'and',  'John']

a = [ 1, 2, 3, 4 ]
b = a                          # like pointer in C, it is a reference;
del a[3]
print b                        # [1, 2, 3]
```

WITH TUPLES *WITH ( )*

Tuples are similar to lists, except they are immutable. You can have nested tuples.

```
tup_long          = ( 'robot', 100 , 3.1415, 'John', 1.61803 )
tup_short         = (123, 'John')
tup_str           = "ice", "cream"
tup_nested        = (123, 'John', (456, 'Benjamin') )
print  tup_long[0]        # 'robot'
print tup_long[1:4]       #100, 3.1415, 'John', 1.61803
print 3 in tup_long       # 'John'
for x in  tup_str:
    print  x, "-"         # ice-cream-
tup_new = tup_long + tup_short
del tup_long; del tup_short
```

Note the last example cause tup_long and tup_short being removed; like an object being deleted, or memory pointer being freed.

Being immutable, the following samples are invalid:

e.g.

```
tuple             = ( 'EV3', 359, 3.1415, 'John', 1.61803 )
list              = [ 'NXT', 299 , 3.23, 'john', 70.2 ]
tuple[2]          = 1000    # Invalid syntax with tuple
list[2]           = 1000    # Valid syntax with list
```

Common operations:

```
cmp(tuple1, tuple2)    len(tuple)     max(tuple)      min(tuple)
```

tuple(list) : convert a list to a tuple

See online document for all Built-in Tuples Functions and Methods

BYTEARRAY

Bytearray objects are created with the built-in function bytearray().

The bytearray class is a mutable sequence of integers in the range 0 <= x < 255.

Sample:

```
def doByteArray():
  elements = [100, 2, 5, 50, 255]
  values = bytearray(elements)    # mutable
 # if you use bytes(elements) , values[ ] will become immutable
  values[0] = 5
  values[1] = 0
  print "There are ", len(elements), "elements: " ,
```

```
        for value in values:
            print (value),
        return
Output :    There are  5 elements:  5 0 5 50 255
```

Note:

If you are interesting to view the memory address, look into using class memoryview. E.g.

```
v = memoryview('0123456789')
print v[0:1]          # this syntax will return the starting address of the "v".
```

MAPPING TYPES

WITH DICTIONARY *WITH { 'KEY1' : 'VALUE', 'KEY2' : I-VALUE, ETC }*

Dictionary are similar to lists, except they are two parts with the following restrictions:

a key and a value pair, where key must be unique.

Value: can be any Python object, either standard objects or user-defined objects

Key: immutable.

```
dictVar= {'Name':'Newton', 'Age': 7, 'Gender': 'Unknown'}
print  dictVar['Name'], " is ", dictVar['Age'], "years old."          #  Newton  is 7 years old.
print "...."
dictVar['Age'] = 8;
dictVar['School'] = "PieSchool";
print dictVar                # School : PieSchool
print "...."
del dictVar['Name']
print dictVar                # School : PieSchool
print "...."
dictVar.clear( )             # remove all entries, but reference to dictVar still exists
del dictVar    # delete reference to dictVar, i.e. can no longer use dictVar
```

Output:

```
Newton  is  7 years old.
....
{'School': 'PieSchool', 'Gender': 'Unknown', 'Age': 8, 'Name': 'Newton'}
....
{'School': 'PieSchool', 'Gender': 'Unknown', 'Age': 8}
....
```

See online document for all Built-in Dictionary Functions and Methods

SET TYPES

WITH SETS *WITH SET (IMMUTABLE OBJECTS)*

A set contains an unordered collection of unique and immutable objects. BUT, the set itself is mutable. We can add or remove items from it.

Python implementation of the sets are just like from sets in mathematics, such as union, intersection, symmetric difference etc.

Since it uses hashing algorithm in manipulating a set, it is much faster than using "list".

```
a = [ ] ;       print(type(a))
a = {} ;        print(type(a))
a = set() ;     print(type(a))
```

Output:

```
<type 'list'>
<type 'dict'>
<type 'set'>
```

```
my_set = set( {1, 2, 3, 4} );          print  "initially: ", (my_set)
my_set.add(2.5) ;          print  "after adding: ", (my_set)
my_set.update([2,3,4,0,10])          print  "after updating: ", (my_set)
my_set.update([4,5], {1,6,8})  print  "after updating: ", (my_set)
```

```
initially:  set([1, 2, 3, 4])
after adding:  set([2.5, 1, 2, 3, 4])
after updating1:  set([2.5, 1, 2, 3, 4, 0, 10])
after updating2:  set([2.5, 1, 2, 3, 4, 5, 0, 8, 10, 6])
========
```

About Data type conversion

This is like data casting.

To convert between types, you simply use the type name as a function.

Function	Description
int(x [,base])	Converts x to an integer. base specifies the base if x is a string.
long(x [,base])	Converts x to a long integer. base specifies the base if x is a string.
float(x)	Converts x to a floating-point number.
complex(real [,imag])	Creates a complex number.
str(x)	Converts object x to a string representation.
repr(x)	Converts object x to an expression string.
eval(str)	Evaluates a string and returns an object.
tuple(s)	Converts s to a tuple.
list(s)	Converts s to a list.
set(s)	Converts s to a set.
dict(d)	Creates a dictionary. d must be a sequence of (key,value) tuples.
frozenset(s)	Converts s to a frozen set.
chr(x)	Converts an integer to a character.
unichr(x)	Converts an integer to a Unicode character.
ord(x)	Converts a single character to its integer value.
hex(x)	Converts an integer to a hexadecimal string.
oct(x)	Converts an integer to an octal string.