

DRAFT

Side-by-side: Python and C/C++

Caveat Emptor: A Few Words about this Document

In writing this document, I have tried to provide a quick introduction to Python for programmers with a basic knowledge of C/C++. Since this is written for programmers, basic programming concepts are not explained. While this is formatted to present C/C++ and Python side-by-side, I do not provide a detailed comparison of the two languages. Rather, I am trying to show how similar things are done in each, and to point out some unique features in Python.

I do not go beyond the basics for either C/C++ or Python. My goal is to provide C/C++ programmers enough information to get started writing Python programs. This document is asymmetric: it is for C/C++ programmers learning Python, not for Python programmers learning C or C++. Since my C/C++ skills are a little rusty (and C++ is definitely a moving target) and the Python material was written as I learned Python, you shouldn't consider me an expert in either C/C++ or Python. Some features of Python are used in examples before they are explained, but the meaning should be obvious from the context.

This is not a stand-alone document. In particular, I do not discuss the philosophy behind Python; nor do I describe how to use the Python interpreter. With that in mind, there are additional resources listed at the end of this document that you might want to consult before you actually begin using Python. I assume that you will consult other documentation and delve further into Python as needed and hope you will quickly outgrow this document.

The Side-by-Side Presentation

Throughout this document, the C/C++ code is presented in the left column or columns with the corresponding Python code will be to the right of the C/C++ code. When C and C++ differ, there may be three columns—one for C, one for C++, and one for Python. All code is presented in colored boxes—light blue, darker blue and light green for C, C++, and Python respectively. When describing a feature of Python that has no corresponding feature in C or C++, no C/C++ code will be included. Similarly, when the code in C or C++ isn't germane or is radically different, it is also omitted. Comments and discussion of the code will span the columns or will be to the right of the code. Output from code is placed to the right of the code when germane to the discussion. Courier New is used for code and output.

Python

Basic Syntax

A Simple Program

Here is “Hello World!” in C++ and Python.

```
#include <iostream>
using namespace std;
int main(void)
{
    cout << "Hello World!";
    return 0;
}
```

```
def main():
    print "Hello World!"

main()
```

There are a few things to note with this simple example. Since Python is interpreted, we didn’t really need a program at all. We could have typed

```
print "Hello World!"
```

directly into the interpreter. With both C/C++ and Python we define and invoke the function `main()`. With C/C++ `main()` is called automatically. With Python, we need to explicitly call it. But since we are explicitly calling `main()` in Python, we could have named the function something else. Unlike C/C++, Python programs don’t have to have a `main()` function.

Also, with Python, we didn’t need any libraries for this simple example. At times Python will require libraries such as `math` or `random`, but a little more seems to be built into core Python than in core C/C++. In this example, there are no semicolons at the ends of the lines. Typically, Python interpreters allow semicolons, but they are rarely used. The exception is when several commands are placed on the same line. In Python you can write

```
x = 3; y = 4; z = x + y; print z
```

7 is displayed

Note that the statements are evaluated in order. In general, it is best to avoid using semicolons since they are an interpreter feature than a language requirement.

Format

Continuing with the sample program, notice that Python uses indentation for identifying blocks of code rather than curly braces. This means that you need to be consistent in your use of indentation. You'll need to start in the first column when entering a new expression into the interpreter.

C/C++ is free-format. Code can start anywhere on the line and a line of code can span multiple lines. When a statement ends, it is marked with a semicolon.

```
foo = bar
    + baz;
```

With Python, a line of code usually ends when the line of text ends. A logical line of code can be extended to the next physical line of text by placing a / as the last character on the physical line.

```
foo = bar /
    + baz
```

In some circumstances, (e.g., unclosed parentheses and brackets), Python will infer that a line is continued even if the / is omitted.

```
mylist = [ 1, 2,
... 3, 4]
```

C/C++ delimits blocks with curly braces:

```
while (check)
{
    i = i + 1;
    check = foo(i);
}
i = 0;
```

Python delimits blocks through indentation. Code that is not part of a nested structure or part of a continuation line should begin in the first column. And don't mix tabs and spaces when indenting unless you are looking for problems.

```
while (check):
    i = i + 1
    check = foo(i)
i = 0
```

Comments

```
/*
This is the tradition C comment
which extends over multiple
lines.
*/
```

```
# Python only has single-line
# comments. Just add more
# lines as needed
```

```
// C++ added single-line comments
```

Unlike C/C++, Python supports imbedded comments know as *docstrings* that are not discarded when the code is compiled, but is retained at runtime. These comments are coded as strings at the top of a module file, top of a function, or top of a class statement. Python automatically shifts these into the `__doc__` attribute.

For example, if you load the following code:

```
"""
spam function
a meaningless function
"""
def spam(x):
    return x + 42
```

the interpreter will display

and then enter the following at the interpreter:

```
print __doc__
```

```
spam function
a meaningless function
```

Identifiers

Legal variable names are basically the same in C/C++ and Python. (Length restrictions may vary from compiler to compiler in C/C++.) Variables must start with a letter or underscore, followed by any combination of letters, digits, or underscores. Both languages are case sensitive.

```
datum
foo1
_bar
```

```
datum
spam1
_eggs
```

Like C/C++, Python has some conventions about when to use variables that start with an underscore so you should use such variables with care. For example, those that start and end with two underscores have special meaning to the interpreter in Python. Names that begin with a single underscore aren't imported with wildcard imports, e.g., `from module import *`. Names beginning with two underscores but not ending in two underscores within a class definition are pseudo-private attributes to the class. The name consisting of just the single underscore in the interpreter typically holds the result of the last evaluation.

In C/C++ and Python, keywords are reserved and cannot be used as identifiers. Like C/C++, some of the words that you might hope were reserved including built-in commands like `open` aren't included.

Keywords

Keywords in C

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

Keywords C++ added to C

and	operator
and_eq	or
bitand	or_eq
bitor	private
bool	protected
break	public
catch	register
class	reinterpret_ca
compl	st
const_cast	static_cast
delete	template
dynamic_cast	this
explicit	throw
export	true
false	try
friend	typeid
inline	typename
mutable	using
namespace	virtual
new	wchar_t
not	xor
not_eq	xor_eq

Keywords in Python

and	pass
assert	print
break	raise
class	return
continue	try
def	while
del	yield
elif	
else	
except	
exec	
finally	
for	
from	
global	
if	
import	
in	
is	
lambda	
not	
or	

Variable Declarations

C/C++ use explicit typing. With C/C++ the type of the variable is stored with the identifier. Thus, unless you redeclare an identifier, its type does not change. Any value you assign to the variable must be the correct type or capable of being automatically recast to that type. C/C++ requires all variable be declared by the time they are used:

```
int foo;
foo = 2;
float bar = 3.3;
foo = 2.34;
// 2.34 is recast as an int
foo = "baz"
// illegal since "baz" can't
// be recast as an int
```

Python uses implicit typing. With Python, the type information is stored with the value. Thus, if you assign a new value to a variable, the type of the variable can change dynamically.

Python deduces the variable type by the way it is used:

```
spam = 2
spam = [1, 2, 3]
spam = "eggs"
```

While Python appears to do more for you, if you misspell a variable name in Python or inadvertently change the type of a variable, Python won't signal an error. It will just assume that you wanted a new variable or that you were done with the old variable and are reusing it for something else.

Assignment

```
x = 1;
y = 2;
```

```
x = 1
y = 2
```

Python also allow simultaneous assignments

```
x, y = 1, 2
```

Simultaneous assignment makes it easy to swap values.

```
x, y = y, x
```

```
temp = x;
x = y;
y = temp;
```

With simultaneous assignments, Python is treating both the left and right-hand sides as implicit tuples. The same approach can be taken explicitly with lists.

```
[a, b, c] = [1, 2, 3]
```

(Tuples and lists are discussed later in this tutorial.)

Simple Types

Integer Types

C/C++ has four kinds of integers that vary based on the storage allocated for each.

```
int foo = 1;
long int bar = -2;
short int baz = 3;
unsigned int bang = 4;
```

Python has two kinds of integers. The default integer type is the same as C's long integer). Arbitrarily long integers vary in size. They are denoted explicitly with an L (deprecated) or implicitly when the numerical value used is too large for a plain integer.

```
spam = 1
longspam = 3L
longspam = 327898769835329324232344
```

In this example, conversion to a long integer is automatic. After the first assignment, the variable is a plain integer, but with the second assignment, the variable changes to a long integer.

```
spam = 3
foo = 3278987698353293242323424
```

Decimal, octal, and hex are denoted the same way in each language, i.e. by the leading digit or digits.

```
int x = 255; //decimal
int y = 0377; //octal
int z = 0xff; //hex
```

```
x = 255 #in decimal
y = 0377 #in octal (decimal 255)
z = 0xff #in hex (decimal 255)
```

Floating Point Numbers

C/C++ provides two storage sizes for floating point numbers.

```
float f = 12.34;
double ff = 12.34e50;
```

Python has a single floating point type that corresponds to C/C++'s double.

```
f = 12.34
ff = 12.34e50
```

Imaginary Numbers

Unlike C/C++, Python includes imaginary numbers as a core type.

```
spam = 2 + 3j
realSpam = spam.real
imaginarySpam = spam.imag
```

You must use a leading integer coefficient with the imaginary part. The operators `.real` and `.imag` extract the real value and the coefficient for imaginary part of the number.

Boolean Type

Although not originally included in either C++ or Python, Boolean variables are now a part of each language.

```
bool x, y;
x = true;
y = false;
```

```
x = True
y = False
```

Note the difference in case

As with C and C++, Python will interpret a zero as false and any other number as true in a conditional expression.

(Strictly speaking, Boolean variables in Python are another integer type.)

```
if (1) printf("true");
if (0.001) printf("true");
if (0) printf("false");
```

```
if (1): print "true"
if (0+1j): print "true"
if (0): print "false"
```

true
true
no output is displayed

(In general, it is best to avoid tests like these.)

None

In Python, the special object called **None** can be thought of as a type with only one element. It is typically used as an empty placeholder. **None** has a truth value of false. It is often used as a place holder in Python.

```
newList = [None]*10
```

This creates a list with placeholders for 10 items.

[[Pointers vs. References—Not Yet Written]]

Unlike C/C++, Python does not explicitly provide pointers. However, much of the behavior can be mimicked.

Type conversion

In most programming languages, including C/C++ and Python, implicit type conversions are made automatically in some circumstances. For example, in mixed-mode expressions, typically simpler types are automatically converted to more complex types. If you add an integer and a float, the integer is automatically converted to a float. In Python, as with C/C++, you can explicitly recast types.

```
(int)32.17
(float)2
(long)2
```

```
static_cast<int>(32.17)
static_cast<float>(2)
static_cast<long>(2)
static_cast<bool>("yes")
static_cast<bool>("no")
static_cast<bool>(0)
```

```
int(32.17)
float(2)
long(2)
bool('yes')
bool('no')
bool(0)
bool("")
str(9.0/10.0)
repr(9.0/10.0)
int('43')
int('43.21') # An Error
```

```
32
2.0
2L
True
True
False
False
'0.9'
'0.900000000000000002'
43
A string must be convertible.
```

In Python, **str()** is a more user friendly string conversion while **repr()** more closely matches the internal representation of a value.

Operators and Precedence

Operators and precedence are very similar. In a few cases, core Python includes operators that are available via libraries in C/C++. Python evaluates expression from left to right except that the right-hand side of an assignment is evaluated before the left-hand side. The following table gives precedence from highest to lowest. The descriptions match the Python operators.

C++ Operations	Python Operations	Python Description
[[Add C++ chart]]	(...), [...], {...}, '...' X.attr, X[i], X[i:j], f(X,Y,...) ** +, -, ~ *, /, % +, - >>, << & ^ in, not in, is, is not, <, <=, >, >=, <>, !=, == not and or lambda	tuples, list, dictionary and strings attributes, indexing, slices, function calls exponentiation unary sign, bit complement multiplication, division, modulus addition subtraction bit shifts bit and bit xor bit or membership, identity, relational operators logical not logical and logical or anonymous function definition

Math operators

```

+
-
*
/
%
```

```

+
-
*
/
%
//
**
```

Addition
Subtraction
Multiplication
Division (truncates on integers only)
Remainder
Division with truncation
Exponentiation

The division with truncation operators, `//`, rounds fractions down for both integers and floats.

There has been some discussion to redesign the `/` division operator in Python so that with integer division it coerces the result into a float if necessary rather than taking the floor. For example, `1/2` would return 0.5 rather than 0. You would use `1//2` if you really want 0. While this change might produce a cleaner, more consistent language, it would break a lot of existing code. If you want to force this behavior, you can add `from __future__ import division` to your code. This redefines `/` to coerce results into floats as necessary. For example,

```

from __future__ import division
1/2
```

0.5

Like C/C++, Python supports update operators but adds a few operators to the mix.

```

x += 1;
x -= 2;
x *= 3;
x /= 4;
x %= 5;
```

```

x += 1
x -= 2
x *= 3
x /= 4
x %= 5
x //= 6
x **= 7
```

Addition
Subtraction
Multiplication
Division with truncation
Remainder
Division with truncation
Exponentiation

Others include the bit operations `&=`, `^=`, `<<=`, `|=`, and `>>=`. While Python supports bit operations such as shifts, etc., they are rarely used and not covered here.

In C/C++ and Python, evaluation of the expression to the right of the equal sign is completed before the variable is updated.

```
int x = 2;
x *= 3 + 7;
```

```
x = 2
x *= 3 + 7
```

In both examples, the final value of x is 20, not 13.

Python does not provide unary increment and decrement operators:

In C/C++ you can write

```
++w;
x++;
--y;
z--;
```

These are syntax errors in Python.

Additional math operations and constants are available in the math library. For example,

```
from math import pi
pi
from math import e
e
```

3.1415926535897931

2.7182818284590451

[[Bit Operations—*Not Yet Written*]]

While more commonly used in lower-level languages, Python does support bit operations.

Relational Operators

Relational operators are the same in these languages.

```
x == 1;
x > 2;
x >= 3;
x < 4;
x <= 5;
x != 6;
```

```
x == 1
x > 2
x >= 3
x < 4
x <= 5
x != 6
```

Equal
Greater than
Greater than or equal to
Less than
Less than or equal to
Not equal to

Like C/C++, these comparisons will return either 0 or 1 which, in a testing situation, will be treated as false or true respectively.

Unlike C/C++, Python does allow simple range testing.

In C/C++, "OK" is printed. This is not what was meant.

```
int x = 0;
if (1 < x < 3) printf("Ok");
```

In Python, nothing is printed which is probably what you want.

```
x = 0
if 1 < x < 3: print "Ok"
```

Also, in Python, statements can't be used as expressions.

Legal C/C++ code—x is reassigned.

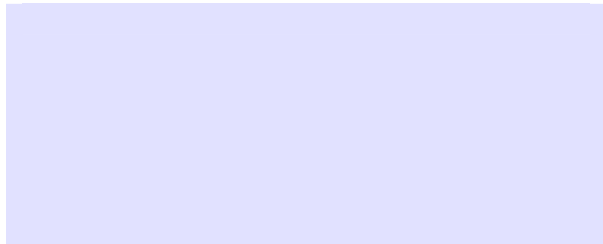
```
if (x = 1) printf("Ok");
```

This is an error in Python

```
if 1 = 3: print "Ok"
```

So, for example, when doing file I/O in Python, you won't be able to combine fetching a value and testing for an end-of-file condition in one operation. But while Python is a little less versatile in this respect, this eliminates a number of very common errors found in C/C++ code.

Python also allows distinguishes between equivalent objects and identical objects with the **is** and **is not** operations.



```
st1 = "Welcome to my world!"
st2 = "Welcome to my world!"
st3 = st1
st3 == st1      True
st3 == st2      True
st3 is st1      True
st3 is st2      False
st3 is not st2  True
```

Note, because Python caches short strings, this doesn't always work the way you might expect it to.



```
st1 = "a"
st2 = "a"
st1 is st2      True
```

Comparisons are applied recursively to data structures.

Logical Operators

Functionally, the same logical operators are available in C/C++ and Python, but different symbols or identifiers are used.

```
(done && found)
(done || found)
(! done)
```

```
(done and found)
(done or found)
(not done)
```

logical and
logical inclusive or
negation

Like C/C++, Python supports short-circuit evaluation. In Python, these `and` and `or` return either the left or right item. A common idiom in Python is to use this to select items.

```
x = 2 or 3
print x
x = 2 and 3
print x
```

2
3

Printing

Both C and C++ use libraries for I/O. For C, the `printf` family of commands found in `stdio.h` are typically used. In C++ these have been largely superseded with stream operations found in `iostream` such as `cout` and `cin`. Python's print commands are much closer to those of C than those of C++.

Libraries

For C, use `stdio.h`

```
#include <stdio.h>
```

You do not need to access external modules when printing in Python

For C++, use `iostream.h`

```
#include <iostream>
```

Special Characters

Escaped characters begin with a `\` and have a special meaning. For example, `\n` is a new line.

```
printf("one\ntwo\nthree");
```

```
print "one\ntwo\nthree"
```

```
one
two
three
```

Here is a list of special characters.

```
\a
\b
\f
\n
\r
\t
\v
\\
\'
\"
\?
```

```
\a
\b
\f
\n
\r
\t
\v
\\
\'
\"
\newline
```

```
alert or bell
backspace
formfeed
newline
carriage return
horizontal tab
vertical tab
backslash
single quote
double quote
question mark
when followed by a newline, \
denotes a continuation line in a
string
```

In addition to these, both C/C++ and Python use `/0` to denote the *null* character. (However, Python does not use the *null* character to terminate strings.)

C/C++ will ignore the backslash for unrecognized escaped characters while Python will print both the backslash.

This displays `:!:`.

```
printf(":\!:"");
```

This displays `:\!:`

```
print ":\!:""
```

You can also use an escape sequence to specify any other character.

```
\ooo
\xhh
```

```
\ooo
\xhh
\uhhh
\Uhhh
\N{id}
```

character code in octal
character code in hex
Unicode 16-bit hex
Unicode 32-bit hex
Unicode dbase id

For example, the ASCII character code for @ is 100 in octal and 40 in hex.

```
printf("\100\x40");
```

```
print "\100\x40"
```

@@

Formatting

In C or C++ you need to explicitly include spaces between item or indicate a linefeed is needed.

```
printf("Hello World\n");
cout <<"Hello" << " " << "World" << endl;
```

Print automatically adds spaces between items and will automatically append a linefeed.

```
print "Hello", "World"
```

Hello World

If you don't want a space, you can build a string.

```
print "Hello"+"World"
print "Hello" "World"
```

HelloWorld
HelloWorld

If you want to continue on the same line, end the print statement with a comma.

```
if 1:
    print "Hello",
    print "World"
```

Hello World

Formatting the output in C

```
printf("int or decimal: %d", 100);
printf("float: %f", 12.34);
printf("formatted int: %5d", 100);
printf("formatted float: %5.2f",
       12.3456);
```

You do not need to access external modules for Python

```
print "int or decimal: %d" % 100
print "float: %f" % 12.34
print "formatted int: %5d" % 100
print "formatted float: %5.2f" % \
      12.3456
```

output

int or decimal: 100
float: 12.340000
formatted int: 100
formatted float: 12.35

Note that the % operator is overloaded for strings.
 This form of substitution can be done in general even if you aren't printing. (See the section on strings.)

```

"%d+%d=%d" % (1, 1, 2)          '1+1=2'
    
```

The general format for a code is `%[flag][width][.precision]code`. For example, `%+8.2f` could be used to specify printing a float within an eight field including the appropriate sign with a precision of two decimal places. (See reference manuals for additional flags.) Here are the common formatting codes with examples:

<code>%c</code>	<code>%c</code>	<i>character</i>	<code>printf(" %c ", 'a');</code>	<code>print "%c" % 'c'</code>	a
<code>%d</code>	<code>%d</code>	<i>decimal integer</i>	<code>printf(" %5d ", 23);</code>	<code>print " %5d " % 23</code>	23
<code>%e</code>	<code>%e</code>	<i>float in exponent form</i>	<code>printf(" %5e ", 0.0005);</code>	<code>print " %5e " % 0.0005</code>	5.000000e-04
<code>%E</code>	<code>%E</code>	<i>float in exponent form with uppercase E</i>	<code>printf(" %5.3E ", 0.0005);</code>	<code>print " %5.3E " % 0.0005</code>	5.000E-04
<code>%i</code>	<code>%i</code>	<i>integer</i>	<code>printf(" %i ", 123);</code>	<code>print " %i " % 123</code>	123
<code>%f</code>	<code>%f</code>	<i>floating point decimal</i>	<code>printf(" %5.2f ", 0.00005);</code>	<code>print " %5.2f " % 0.00005</code>	0.00
<code>%g</code>	<code>%g</code>	<i>floating point e or f</i>	<code>printf(" %5g ", 0.00005);</code>	<code>print " %5g " % 0.00005</code>	5e-05
<code>%G</code>	<code>%G</code>	<i>floating point E or f</i>	<code>printf(" %5.2G ", 0.00005);</code>	<code>print " %5.2G " % 0.00005</code>	5E-05
<code>%p</code>	<code>%p</code>	<i>pointer</i>	<code>printf(" %p ", p1);</code>		0xc0000000
	<code>%r</code>	<i>string (using repr() in Python)</i>		<code>print " %r " % "hello"</code>	'hello'
<code>%s</code>	<code>%s</code>	<i>string (using str() in Python)</i>	<code>printf(" %s ", "hello");</code>	<code>print " %s " % "hello"</code>	hello
<code>%u</code>	<code>%u</code>	<i>unsigned</i>	<code>printf(" %u ", 123);</code>	<code>print " %u " % 123</code>	123
<code>%o</code>	<code>%o</code>	<i>octal integer</i>	<code>printf(" %o ", 04);</code>	<code>print " %o " % 04</code>	4
<code>%x</code>	<code>%x</code>	<i>hex integer</i>	<code>printf(" %x ", 0xff);</code>	<code>print " %x " % 0xff</code>	ff
<code>%X</code>	<code>%X</code>	<i>hex integer with uppercase</i>	<code>printf(" %X ", 0xff);</code>	<code>print " %X " % 0xff</code>	FF
<code>%%</code>	<code>%%</code>	<i>literal "%"</i>	<code>printf("10%%");</code>	<code>print "%i%" % 10</code>	10%

Input

For simple input in Python, you can use `input()` or `raw_input()`.

For C++, the target variable must be of the appropriate type.

```
cout <<"Type something: ";  
cin >> x;
```

With `input()`, the argument is the prompt and whatever the user types is the returned value.

```
x = input("type something: ")
```

`raw_input()` assumes the input data is a string.

```
x = raw_input("type something: ")
```

Of course there are other options, particularly with C/C++, but we won't go into those here.

[[File I/O—Not yet Written]]

Aggregate Data Types

One area the C/C++ and Python deviate greatly is in the area of aggregate data types.

Character and Strings

Character Data

While C/C++ have a separate data type for characters, Python does not. In Python, you'll need to use strings of a length of one whenever you want to mimic the behavior of C/C++ character data. In practice, this will not be much of a problem.

Strings

In C, strings are treated as an array of characters that is terminated by a null character. C++ introduces a more powerful string class made available by the inclusion of the appropriate library. While Python provides much of the functionality associated with the C++ string class, superficially at least, strings in Python more closely resemble those of C. However, they have at least two key crucial differences. First, strings in Python are not terminated with a null character. This

means you won't be able to (or need to) iterate over a sting looking for its end. Perhaps a more significant difference is that strings in Python are immutable. You can't directly change strings in Python. Rather, when you need to "change" a sting in Python, you will construct a new string based on the original.

For C, strings are just arrays of characters

For C++, you'll need to include the library
string

```
#include <string>
```

For Python, strings are an intrinsic type

In some contexts, such as printing, you can use string literals as constants without declaring them

```
printf("Hello");
```

```
cout << "Hello";
```

```
print "Hello"
```

Usually, however, you'll need to declare and initialize your strings.

```
char s[] = "Once upon a time";
char s[] = "";
char s[4] = {'C', 'a', 't',
            '\0'};
```

```
string s = "Once upon a time";
string s1 = "";
```

```
s = "Once upon a time"
s = ""
s = 'Cat'
```

In the second line we created empty strings. Notice the use of single quotes in the last line. Single quoted delimit characters in C. Since characters aren't a separate type in Python, either single or double quotes can be use to delimit strings. There are some other ways to delimit strings in Python. Having two different delimiters for strings in Python make it easier to specify strings with embedded quotes. I

You can delimit strings with triple quotes in Python. Triple quotes allow you to include newlines in the text. But you could also use escape sequences to get the same results.

```
s1 = '''Hello'''
s2 = """Hello
Sailor"""
```

```
'Hello'
'Hello\n\nSailor'
```

Raw strings ignore escape sequences. Compare the following:

```
s3 = "\\t\\"
print s3
s4 = r"\\t\\"
print s4
```

```
\    \
\\t\\
```

	Unicode strings include Unicode characters. s5 = u'spam\u0020eggs' print s5 s1[1] = 'a'	spam eggs
--	--	-----------

Note that the last example is illegal in Python. You cannot change part of a string. However, you can easily construct new strings.

	s1 = s1[:1]+'a'+s1[2:] print s1	'Hallo'
--	------------------------------------	---------

This new construct makes use of string slices, described in the next section.

String Slices

In C you typically work with one character at a time. If you want more functionality, you link to the appropriate library. C++ provides methods through the string class that provide even greater functionality. With Python, this is all built-in. Indexing is from zero in C/C++ and Python.

In C you typically work with one character at a time without bounds checking char s[] = "Hello sailor"; printf("%c\n", s[0]);	With C++, the at method provides bounds checking. string s = "Hello sailor"; printf("%c\n", s[1]); printf("%c\n", s.at(1));	Python also provides bounds checking. s = "Hello Sailor" print s[0]
In C, this produces garbage. printf("%c\n", s[20]);	In C++, this generates an exception. printf("%c\n", s.at(20));	In Python, this generates an exception. print s[20]

Using a technique known as slices, Python provides a easy mechanism to work with parts of strings. Slices allow you to specify a range within a string and Python returns a new string constructed from that range. Remember, Python strings are immutable so you must create copies.

	Indexing starts at 0 and goes to one less than the length of the string. The character at the first index is included but the slice stops just before the last indexed character. s1 = "Hello Sailor" s1[0:5] s1[1:2]	'Hello' 'e'
--	--	----------------

If an index is missing, it defaults to the corresponding end of the string. (The last example gives one way of copying a sting.)

```
s1[:5]      'Hello'
s1[6:]     'Sailor'
s1[:]      'Hello Sailor'
```

Negative numbers can be used to count from the end of the string.

```
s1[-2:]    'or'
s1[-6:-2]  'Sail'
```

A third argument can be used as a step size

```
s1[0:5:2]  'Hlo'
s1[: :2]   'HloSio'
```

As show above, slices with empty limits, e.g., `s1[:]`, simply copy the original string. This is an extremely common idiom in Python.

Scanning over strings is often done with a `for/in` construct.

```
s1 = 'cat'
>>> for char in s1:
      print char
```

Displays

```
c
a
t
```

String Operations and Methods

Python provides a number of operations and methods for working with strings. Similar functions are available via libraries in C and C++. Since these aren't part of the core C/C++ languages, they aren't include here.

Finding the length of a string

```
len(s1)
```

12

Repeating stings

```
'Ni' * 4
```

'NiNiNiNi'

	Concatenating strings <code>'Ni' + 'Ni'</code> <code>'Ni''Ni'</code>	<code>'NiNi'</code> <code>'NiNi'</code>
	Searching—optional parameters say where to start and stop the search. <code>s1.find('Sail')</code> <code>s1.find('lo', 6)</code> <code>s1.find('lo', 6, 8)</code>	<code>6</code> <code>9</code> <code>-1 (Not found.)</code>
	Replacing <code>s2 = s1.replace('S', 's')</code> <code>s1</code> <code>s2</code> <code>s1.replace('l', 'x', 2)</code>	<code>'Hello Sailor'</code> <code>'Hello sailor'</code> <code>'Hexxo Sailor'</code>
	Splitting <code>s1.split()</code> <code>s1.split('l')</code>	<code>['Hello', 'Sailor']</code> <code>['He', '', 'o Sai', 'or']</code>
	Membership <code>'l' in s1</code> <code>'x' in s1</code>	<code>True</code> <code>False</code>
	String Conversion <code>str(3.14)</code> <code>float('3.14')</code> <code>int('42')</code>	<code>'3.14'</code> <code>3.140000000000000001</code> <code>42</code>
	Exploding and Imploding Strings <code>[A, B, C] = 'abc'</code> <code>A</code> <code>''.join([C, B, A])</code> <code>(A, B, C) = '123'</code> <code>A</code> <code>''.join((A,B,C))</code>	<code>'a'</code> <code>'cba'</code> <code>'1'</code> <code>'123'</code>

Here is a incomplete but longer list of Python methods.

```
capitalize
center
count
endswith
expandtabs
find
index
isalnum
isalpha
isdigit
islower
isspace
istitle
isupper
join

ljust
lower
lstrip
replace
rfind
rindex
rjust
rstrip
split
splitlines
startswith
strip
swapcase
title
upper
zfill
```

```
'hello'.capitalize()
'spam'.center(12)
'butterfly'.count('t')
'butterfly'.endswith('fly')
'x\tx'.expandtabs()
'rabbit'.find('bit')
'rabbit'.index('b')
'x'.isalnum()
'7'.isalpha()
'7'.isdigit()
'H'.islower()
'H '.isspace()
'The Book'.istitle()
'H'.isupper()
'ABC'.join('xxx')
''.join(["1", "a", "A"])
'Start'.ljust(10)
'ABC'.lower()
' X X '.rstrip()
'ham'.replace('h', 'sp')
'OTTO'.rfind('O')
'Otto'.rindex('t')
'End'.rjust(10)
' X X '.rstrip()
'How now'.split('o')
'How\nnow'.splitlines()
'Butterfly'.startswith('fly')
' X X '.strip()
'Hello'.swapcase()
'the book'.title()
'abc'.upper()
'.1'.zfill(4)
```

```
'Hello'
'  spam  '
2
True
'x      x'
3
2
True
False
True
False
False
True
True
True
'xABCxABCx'
'laA'
'Start  '
'abc'
'X X  '
'spam'
3
2
'      End'
' X X'
['H', 'w n', 'w']
['How', 'now']
False
'X X'
'hELLO'
'The Book'
'ABC'
'00.1'
```

Additional arguments are supported for some of these methods.

Lists

Lists vs. arrays

Where C/C++ provides arrays and structures, Python provides list, tuples, and dictionaries. Unlike arrays, lists, tuples, and dictionaries can contain heterogeneous data. We will begin by contrasting arrays and lists.

Arrays must be explicitly declared in C/C++ and must contain homogeneous data.

```
int dat[5];
float L[] = {1.1, 2.2, 3.3};
```

In Python, lists are defined when first used.

```
dat = [0, 1.1, 'spam', 42]
L = [1.1, 2.2]
```

The length of an array is fixed in C/C++.

Lists are dynamic in Python

```
L.append(3.3)
```

L is now [1.1, 2.2, 3.3].

Arrays are indexed from 0.

```
y = L[0];
```

Lists are indexed from 0.

```
y = L[0]
```

y is now 1.1.

Arrays are mutable.

```
L[0] = 4.4;
```

Lists are mutable.

```
L[0] = 4.4
```

L is now [4.4, 2.2, 3.3].

There are no bounds checking in C/C++. Illegal accesses produce unpredictable results.

```
y = L[10]; // garbage value in y
```

Python generates an exception with an illegal access.

```
y = L[10] # throws an exception
```

Multidimensional arrays are supported.

```
int dat[2][3]={{1, 2, 3},
              {4, 5, 6}};
printf("%d\n", dat[0][0]);
printf("%d\n", dat[1][2]);
```

Multidimensional arrays are supported.

```
dat=[[1, 2, 3], [4, 5, 6]]
```

```
print dat[0][0]
print dat[1][2]
```

1
6

(The C++ vector class was created to address some of these restrictions but is not addressed here since it is not part of the core C++ language.)

Lists vs. structures

There are also some similarities between lists and structures.

Structures must be explicitly declared in C/C++. While they can contain mixed data, their format is fixed.

```
struct student {
    char * name;
    int age;
    float gpa; }
s1, s2, s3;
```

You need to redefine a structure if you want something different

```
struct student {
    char * name;
    int age;
    float gpa;
    char * birthplace}
s1, s2, s3;
```

Structures are accessed by name.

```
s1.age = 63;
```

Structures are mutable.

```
s1.age++;
```

In Python, lists are defined when first used and their format can vary dynamically.

```
s1 = ['Eric Idle', 63, 4.0]
```

Lists are dynamic in Python

```
s1.append('South Shields')
```

Lists are accessed by indices. (If you need to access data by name, use a Python dictionary.)

```
s1[1] = 63;
```

Lists are mutable.

```
s1[1] += 1
```

In C++, structures can be nested

```
struct student {
    char name[100];
    int age;
    float gpa; } s1, s2, s3;

struct classlist {
    student
        aStudent;
    student
        anotherStudent; } myClass;
```

Lists can be nested

```
s1 = ['Eric Idle', 63, 4.0]
s2 = ['John Cleese', 67, 4.0]
myClass = [s1, s2]
```

List functions in Python

Python provides a number of built-in functions for working with lists. Here are a few:

Concatenate

```
f0 = []
f1 = ['ham', 'eggs']
f2 = ['eggs']
f1 + f2
```

An empty list.

```
['ham', 'eggs']
['eggs']
['ham', 'eggs', 'eggs']
```

Repeat

```
f3 = [1, 2]
f3 * 2
```

```
[1, 2, 1, 2]
```

Append

```
f1.append('spam')
```

```
['ham', 'eggs', 'spam']
```

Sort

```
f1.sort()
```

```
['eggs', 'ham', 'spam']
```

```
f3.extend(f3)
```

```
[1, 2, 1, 2]
```

	Slices <code>f1[1:3]</code>	<code>['ham', 'spam',]</code>
	Search <code>f1.index('spam')</code>	<code>2</code>
	Reverse <code>f1.reverse()</code>	<code>['spam', 'ham', 'eggs']</code>
	Delete <code>del f1[2]</code>	<code>['spam', 'ham']</code>
	Remove and return <code>f1.pop()</code> <code>print f1</code>	<code>'ham'</code> <code>['spam']</code>
	List length <code>len(f1)</code>	<code>1</code>

Scanning over lists is typically done with a **for/in** construct.

	<code>L1 = [1,2,3,4]</code> <code>for item in L1:</code> <code> print item</code>	<code>1</code> <code>2</code> <code>3</code> <code>4</code>
--	--	--

Tuples

Python tuples are very similar to Python list except tuples are immutable and are denoted with parentheses rather than square brackets.

	<code>f0 = ()</code> <code>f1 = (1, 2, 3)</code>	<i>An empty tuple</i> <code>(1, 2, 3)</code>
--	---	---

While most list operations won't work on tuples, some work by returning a new tuple. In each case the original tuple is unchanged. These are limited to the following

	Concatenate <code>f1 + f1</code>	<code>(1, 2, 3, 1, 2, 3)</code>
	Repeat <code>f1 * 2</code>	<code>(1, 2, 3, 1, 2, 3)</code>
	Slices <code>f1[1:3]</code>	<code>(2, 3)</code> <i>Note the use of square braces when indexing tuples.</i>
	Length <code>len(f1)</code>	<code>3</code>

As with strings, when you need a different tuple, you create a new tuple based on the original tuple rather than change the existing tuple.

One word of warning, while tuples are immutable, that only applies to the highest level. Objects within tuples may be changed.

	<pre>L1 = [1, 2, 3] T1 = (7, L1, 13) L1[1] = 'pig' T1</pre>	<code>(7, [1, 'pig', 3], 13)</code>
--	---	-------------------------------------

Tuples are often used to “create” enumerated types in Python.

<pre>enum season {winter, spring, summer, fall} s1;</pre>	<pre>(winter, spring, summer, fall) = \ range(4)</pre>
--	--

Dictionaries

Dictionaries, sometimes called associative arrays in other programming language, allow you to create and access a collection of data using keys rather than indices. Like lists, dictionaries are mutable collections.

Dictionaries are enclosed in curly braces and consist of *key:value* pairs.

```
f0 = {}
d1 = {'fst':1,
      'mid':'spam',
      'lst':[1,2] }

d1['fst']
```

An empty dictionary
A three item dictionary

1

Dictionaries can be nested

```
d2 = {'meat' : {'good':'ham',
               'better':'spam'},
      'bread' : {'good':'rye',
                 'better':'wheat'}}

d2['meats']['better']
```

'spam'

Be warned, since you are accessing a dictionary by key and not by position, Python may store the pairs in a different order.

Python provides a number of methods for working with dictionaries. Here are a few:

Dealing with keys

```
d1 = {'fst':1,
      'mid':'spam',
      'lst':[1,2] }

d1.has_key('mid')
d1.has_key('spam')
d1.keys()
```

True
False
['lst', 'fst', 'mid']

You can also get a list of values.

```
d1.values()
```

[[1, 2], 1, 'spam']

You have a couple of options for accessing values in a dictionary. You can use brackets in the same way you would with other collections or you can use the `get` method which has the advantage of allowing you to specify a default value.

```
d1['mid']
d1.get('mid', 42)
d1.get('spam', 42)
```

'spam'
'spam'
42

The length of a dictionary is the number of pairs in it.

```
len(d1)
```

3

If you have lists of keys and values or tuples of key/value pairs, you can convert them into dictionaries using `zip()` and `dict()`.

```
myKeys = ['P', 'C']
myValues = ['s', 'f']
myTuples = zip(myKeys, myValues)
myTuples
myDictionary = dict(myTuples)
myDictionary
```

[('P', 's'), ('C', 'f')]
{'P': 's', 'C': 'f'}

You can also copy dictionaries when needed.

```
newDictionary = myDictionary.copy()
newDictionary
```

{'P': 's', 'C': 'f'}

Control Structures

It is the ability to make decisions that distinguishes a computer from a calculator.

if-else

With most languages, while loops are the most general iterative statement. The general format for a while statement is basically the same in C/C++ and Python except for a couple of syntactic differences.

```
if (test)
{
    body1
}
else
{
    body2
}
```

```
if test:
    body1
else:
    body2
```

In Python,

- i. you don't need parentheses around the test but you do need the colon*
- ii. the body is indented rather than enclosed in curly braces*
- iii. you use `else:` rather than `else`.*

The `else`-clause is optional in C/C++ and Python.

As with C/C++, you can do a series of test:

```
if (test1)
{
    body1
}
else if (test2)
{
    body2
}
```

```
if test1:
    body1
elif test2:
    body2
```

Notice that Python uses `elif` rather than `else if`.

Python does not have a multi-way branch statement like C/C++. In practices, this isn't much of a problem. You can often mimic a switch-statement using dictionaries and lambda functions.

```
switch (x)
{
  case 1 :
    printf("Mon\n");
    break;
  case 2 :
    printf("Wed\n");
    break;
  case 3 :
    printf("Fri\n");
    break;
  default :
    printf("Free\n");
}
```

```
print {1:'Mon',
      2:'Tue',
      3:'Wed'}[x]
```

while-loops

With most languages, while loops are the most general iterative statement. The general format for a while statement is basically the same in C/C++ and Python except for a couple of syntactic differences.

```
while (test)
{
  body
}
```

```
while test :
  body
```

In Python:

i. you don't need parentheses around the test but you do need the colon

ii. the body is indented rather than enclosed in curly braces

Here is an example:

```
int i = 5;
while (i >= 0)
{
  printf("%d\n", i);
  i--;
}
```

```
i = 5
while i >= 0 :
  print i
  i = i - 1
```

5
4
3
2
1

}

0

With both C/C++ and Python, you can use **break** to exit from a loop and **continue** to jump back to the test

Using **break**

```
int i = 5;
while (1)
{
    printf("%d\n", i);
    i--;
    if (i == 0) break;
}
```

```
i = 5
while 1 :
    print i
    i = i - 1
    if (i == 0) : break
```

5
4
3
2
1

Using **continue**

```
int i = 6;
while (i > 0)
{
    i--;
    if (i % 2 == 0)
        continue;
    printf("%d\n", i);
}
```

```
i = 6
while i > 0 :
    i = i - 1
    if (i % 2 == 0) :
        continue
    print i
```

5
3
1

In addition, in Python you have the option of adding an else clause to a while loop. The general format is

```
while test1 :
    body1
    if test2 : break
else:
    body2
```

If you exit the loop normally (because test1 fails), then body2 will be executed. If exit loop by way of the break (test2 succeeds), you will skip body2.

for-loops

for-loops are more versatile in Python than in C/C++ because they are designed to iterate over all collections of objects. Consider string. With C/C++, you need to create an index to access each character in a string. With Python, you can simply iterate over the string itself.

i is an index for each letter in *s*.

```
for (i = 0; i < 5; i++)
    if (s[i] == 'l')
        printf("Found an 'l'\n");
```

letr is set to each letter in *s*.

```
for letr in s:
    if (letr == 'l'):
        print "Found an 'l'"
```

If *s* is "hello", the output is

Found an 'l'
Found an 'l'

This approach works for any collection in Python.

If you need numerical values in Python, you can use `range` to create a list of integers and then iterate over the list. Here is an example of how `range` works:

	5 items starting at 0 <code>range(5)</code>	[0, 1, 2, 3, 4]
	start at 5, stop before 10 <code>range(5,10)</code>	[5, 6, 7, 8, 9]
	start at 0, count by 2 <code>range(0,10,2)</code>	[0, 2, 4, 6, 8]
	count down by 7 <code>range(100,75,-7)</code>	[100, 93, 86, 79]

Here is the count down example again. Notice that we've used `range` inside the for-loop. (Can you explain the parameters to `range`?)

```
for (i=5; i>=0; i--)
    printf("%d\n", i);
```

```
for i in range(5,-1,-1):
    print i
```

5
4
3
2
1
0

As with the while loop, with both C/C++ and Python, you can use **break** to exit from a loop and **continue** to jump back to the test. In addition, in Python you have the option of adding an else clause to a **for**-loop. The general format is

```
for item in object:
    body1
else:
    body2
```

```
for item in object:
    body1
    if test : break
```

If you exit the loop normally), then body2 will be executed. If exit loop by way of the break (test

```
else:
    body2
```

succeeds), you will skip body2.

Functions

“The typical way to get something done in a C++ program is to call a function to do it.”—Bjarne Stroustrup

Functions in Python are quite similar to functions in C/C++ but there are a few significant differences. Here is an example of a simple function definition:

First the prototype

```
int square(int);
```

Python doesn't use prototypes

Now the definition

```
int square(int x)
{
    return x * x;
}
```

```
def square(x):
    return x * x
```

Note the keyword `def` and the use of the colon in Python.

Finally, using the function

```
y = square(3);
printf("3 squared is %d.\n",
       square(3));
```

```
y = square(3)
print "3 squared is %d.\n" % \
      square(3)
```

Of course, you can have many, many lines of code in the body of the function before the return.

As with C/C++, you can write ersatz procedures by omitting the return statement.

First the prototype

```
void display_sq(int);
```

Again, no prototype in Python

Now the procedure definition

```
void display_sq(int x)
{
    printf("The square of %d is
%d.\n",
          x, square(x));
}
```

```
def display_sq(x):
    print "The square of %d is %d."
    \
      % (x, square(x))
```

Finally, calling a procedure

```
display_sq(3);
```

```
display_sq(3)
```

With Python, def executes at runtime and can appear anywhere a statement can occur.

```
if x < 0:
    def to_zero(y):
        y += 1
        return y
    x = to_zero(x)
else:
    def to_zero(y):
        y -= 1
        return y
    x = to_zero(x)
```

While there are much simpler ways to do this, this will add 1 to x if it is negative or subtract 1 if x is positive. The function to_zero() still be available after this block of code executes.

Because Python does not specify the type of the parameters, its functions are inherently polymorphic.

```
def triple(x):
    return x * 3

triple(5)          # 5 * 3
triple(5.2)        # 5.2 * 3
triple('Ni ')     # 'Ni ' * 3
```

```
15
15.6
'Ni Ni Ni'
```

Passing variables

Pass-by-value

```
#include <stdio.h>
void foo(int);

int main ( void )
{
    int x = 5;
    foo(x);
    printf("%d\n", x);
    return 0;
}
```

```
def spam(x):
    x += 1

x = 5
spam(x)
```

x is unchanged.

```

}
void foo(int x)
{  x++; }

```

Pass-by-reference in C++. (In C you would use pointers.)

```

#include <iostream.h>
using namespace std;

void foo(int &);

int main ( void )
{
    int x = 5;
    foo(x);
    printf("%d\n", x);
    return 0;
}

void foo(int & x)
{  x++; }

```

C/C++ default to pass-by-reference with arrays

```

#include <stdio.h>

void foo(int[]);

int main ( void )
{
    int x[] = {1,2,3,4,5};
    foo(x);
    printf("%d\n", x[0]);

    return 0;
}

void foo(int x[])

```

With Python, any mutable collection can be altered in a function

```

foods = ['ham', 'eggs']
def spam_it(L):
    L[0] = 'spam'

spam_it(foods)

```

This isn't an option with Python. But there is a simple way around this explained below.

foods is changed to ['spam', 'eggs']

While this works with lists and dictionaries, it would fail with tuples and strings since tuples and strings are immutable collections.

```
{
    x[0]++;
}
```

If you wish to avoid making changes to mutable objects, you can make explicit copies by using slices.

```
spam_it(foods[:])
```

spam_it is given a copy of foods in this example.

Like C++, Python functions support default values for arguments and can have an unspecified number of arguments. Using default values is straightforward:

```
#include <iostream.h>
using namespace std;

int foo(int , int = 10);

int main ( void )
{
    printf("%d\n", foo(5));
    printf("%d\n", foo(5, 5));
    return 0;
}

int foo(int x, int y)
{ return x + y; }
```

```
def spam(x, y=10):
    return x + y

spam(5)
spam(5, 5)
```

15
10

Dealing with an unspecified number of arguments can be awkward.

With C++, a set of macros in `<stdarg>` can be used for accessing unspecified arguments.

```
int printf(const char* ...);
```

With Python, unspecified arguments are simply packed in a tuple or dictionary.

```
def func(*name1)
def func(**name2)
```

name1 gets unspecified positional arguments.
name2 gets unspecified key-work arguments.

Python also supports keyword arguments. To use keywords, you don't have to define your functions differently. You just need to know the names of the parameters.

```
def breakfast(spread, bread):
    print "Put the " + spread + \
        " on the " + bread + "."

breakfast('jelly', 'rye')

breakfast(bread='rye',
          spread='jam')

breakfast('butter', bread='bagel')

breakfast(spread='butter', 'rye')
```

When using positional arguments, the order is essential.
When using keyword arguments, order doesn't matter.
When using both, the positional arguments must come before the keyword arguments.
This is incorrect.

In Python, you can combine default values, keyword, and an unspecified number of arguments as needed.

Returning values

On the surface, return appears to work the same in C/C++ and Python.

```
int five(void)
{   return 5;   }
```

```
def five():
    return 5
```

Both return the integer 5.

But because Python can return tuples and supports simultaneous assignments, we can return multiple values in Python.

```
def get_info():
    name = raw_input("Name: ")
    addr = raw_input("Address: ")
    phone = raw_input("Phone no: ")
    return (name, addr, phone)

n1, a1, p1 = get_info()
```

This code prompts the user for their name, address, and phone number returning all three as a tuple.

The three returned values are

stored in the variable `n1`, `a1`, and `p1`.

By either passing values in a list or by returning multiple values as tuples, it is straightforward to “simulate” pass-by-reference in Python.

You can also return values in Python using a **yield** statement. But unlike **return**, **yield** allows you to start over where you left off.

```
def integersTo(n):
    for i in range(n):
        yield i
ints = integersTo(10)
ints.next()
ints.next()
ints.next()
ints.next()
for j in ints: print j,
```

0
1
2
3
4 5 6 7 8 9

This allows the creation of feature called generators.

Recursion

As with C/C++, Python supports recursion.

```
int fact(int x)
{
    if (x == 1) return 1;
    else return x * fact(x-1);
}
```

```
def fact(x):
    if (x == 1): return 1
    else: return x * fact(x-1)
```

Both return the integer 120.

Scope

While occasionally global variables may be need, in general they should be avoided.

With C/C++, if a variable is not declared within a function, the enclosing scope is searched for the variable.

```
#include <stdio.h>

int other = 5;
int foo(int);

int main ( void )
{
    printf("%d\n",foo(2));
    return 0;
}

int foo(int x)
{
    return x + other;
}
```

With Python, the variable must be explicitly designated as a global if you wish to change its value. Otherwise, while you can still access its value, you won't be able to change that value outside the function

```
other = 5
def spam(x):
    global other
    other = x
    return other + x

spam(2)
```

In both these examples, other is a global variable.

With C/C++, you can not nest function definitions within other functions. This is allowed in Python.



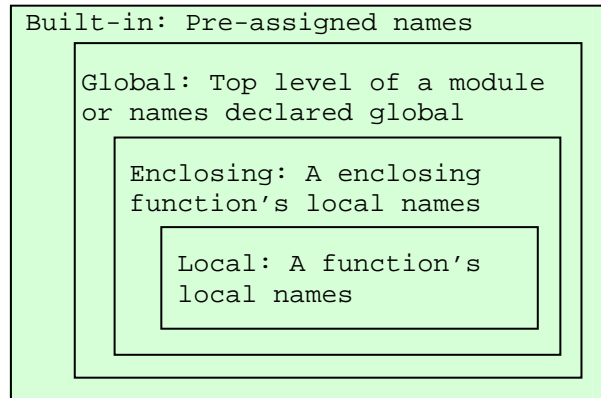
```
def spam(x):
    def eggs(y):
        def grits(z):
            return 2 * z
        return 2 * grits(y)
    return 2 * eggs(x)

spam(2)
```

In this code, eggs() is defined inside spam() and grits() is defined inside eggs(). The user will be able to call spam(), but not eggs() or grits(). The function spam() can call eggs() (or spam()) but not grits().

This would return 16.

The boxes show the enclosing scopes. Functions inside a box can look out but functions outside a box can't peak into a box. The next figure shows scopes in Python.



Functional Programming in Python

While it is certainly possible to pass pointers to functions, etc., in general functional programming is a difficult undertaking with C/C++. Python, on the other hand, provides direct support for functional programming including unnamed or **lambda** function, and functional operators such as **map** and **reduce**. Here are a few simple examples:

For anonymous or unnamed functions, use `lambda()`.

```
(lambda(x):x+2)(3)      5
spam = lambda(x): 2 * x
spam(3)                 6
```

You can use `apply()` to apply a function to a list of arguments.

```
apply(lambda x, y: 2 * x + y, (10, 5))  25
apply(abs, [-5])                        5
```

`map()` is used to repeatedly apply a function to the elements in a sequence.

```
map(abs, (-3, 2, 5, 7))                [3, 2, 5, 7]
map(abs, [-3, 2, 5, 7])                [3, 2, 5, 7]
```

To repeatedly apply a function taking arguments from a sequence, use `reduce()`.

```
reduce(lambda x, y: x + y, [1, 2, 3, 4, 5])  15
from operator import add
reduce(add, [1, 2, 3, 4, 5])                15
```

Use `filter()` to apply a Boolean operator to select items in a sequence.

```
filter(lambda x : x > 0 and x % 2
== 0, range(-5, 15))                    [2, 4, 6, 8, 10, 12, 14]
```

Because mapping over sequences is such a common task, Python provides list comprehensions, a related feature. List comprehensions collect the results of repeated applying an expression to the items in a sequence. Here are a few examples:

```
[2 * x for x in range(10)]  
from math import sin  
[str(round(sin(x), 3)) for x in  
(0.1, 0.2, 0.3)]
```

```
[0, 2, 4, 6, 8, 10, 12,  
14, 16, 18]  
['0.1', '0.199',  
'0.296']
```

[[Object Oriented Programming—Incomplete]]

Like C++, Python supports object orient programming. Here is a partial class definition a class defining rational numbers.

With C/C++,



With Python,

```
class rats:
    def __init__(self, nu, de):
        factor = gcd(nu, de)
        self.num = nu / factor
        self.den = de / factor
    def display(self):
        print self.num, "/",
self.den
    def __add__(self, aRat):
        newNum = self.num *
aRat.den + \
                self.den *
aRat.num
        newDen = self.den *
aRat.den
        factor = gcd(newNum,
newDen)
        newNum /= factor
        newDen /= factor
        return rats(newNum, newDen)

def gcd(m, n):
    while m%n != 0:
        m,n = n, m%n
    return n
```

In both these examples, other is a global variable.

[[Inheritance]]

[[Exceptions—Not Yet Written]]

[[Assert p134, raise, try/except/finally]]

Python Resources

As is standard practice, these really useful resources are listed at the end of the document to make sure you read what I have to say first.

On-line resources:

Guido van Rossum's tutorial: <http://docs.python.org/tut/>

Other Internet sources on Python: <http://www.python.org/doc/>

For information on IDLE, the Python IDE: <http://www.python.org/idle/>

Learn Python in 10 minutes: <http://www.poromenos.org/tutorials/python>

Books you may find useful:

Mark Lutz & David Ascher, *Learning Python*, O'Reilly Media—written for the seasoned programmer learning Python and strongly recommended.

Mark Lutz, *Python Pocket Reference*, O'Reilly Media—as the name says, a pocket reference. Very handy.

Bradley Miller and David Ranum, *Problem Solving with Algorithms and Data Structures using Python*, Franklin, Beedle & Associates—data structures text.

John Zelle, *Python Programming An Introduction to Computer Science*, Franklin, Beedle & Associates—for those new to programming.

Of course, there are many, many other resources.