

# Reading Assignment

- Python Tutorial  
(linked from class web page)

# Scripting Languages

The languages we've seen so far are "complete" programming languages. They are designed to encode a complete program, perhaps with the assistance of library routines.

However, throughout the history of programming another sort of programming language has evolved (and flourished)—the scripting language.

Scripting languages evolved as simple tools to "glue" together existing programs and utilities.

When batch programming (rather than interactive programming) was dominant, *command languages* evolved to program the steps necessary to run a program.

# JCL

One of the most widely used command languages was IBM's *JCL* (Job Control Language). It was designed to "program" a batch operating system. It directed the steps required to put together a program and run it. A JCL program

1. Identified your job to the system.
2. Directed the execution of the program(s).
3. Described the devices (readers, printers, etc.) and data needed by the program(s).

The following example demonstrates the JCL necessary to run an assembler language program. The job's name is

**MYPROG**, and it belongs to Clem Kaddiddlehopper who wants to have the output returned to room 222.

```
//MYPROG JOB, 'CLEM
          KADDIDDLEHOPPER',TIME=(0,5)
/*JOBPARM          ROOM=222
//STEP1 EXEC ASMHCG
//ASM.SYSIN DD *
[assembler source deck]
/*
//GO.DATA DD *
[data deck]
/*
```

# Shell Scripts

By today's standards JCL is crude indeed, but it did establish the notion of a specialized programming language used to "program" the steps of an operating system. These ideas evolved into the far more powerful *shell scripts* used by systems like Unix.

Many utility programs provided by Unix (and other operating systems) are not conventional programs compiled from C or C++. Instead, they are small shell scripts that tie together existing programs into new and useful forms.

Here is a typical Unix shell script:

```
# wl - run word count,  
# then print (lpr) a set of files  
#  
# Usage: wl file1 [file2] ...  
set T=/tmp/wl.$$ # temporary file  
wc -l $* > $T # get line counts  
lpr $T $* # print line counts,  
# then files  
rm $T # remove temporary file
```

Shell scripts are useful because they are more direct and concise than ordinary C or C++ programs.

Experienced Unix programmers prefer to build a new utility from existing ones rather than from scratch. It's much easier, faster, and far less error prone. Other scripting languages, like Python, can be used in shell scripts.

# JavaScript and VBScript

In the HTML that implements web pages it is often necessary to perform simple computations. You could, of course, write a Java applet, and call it, but this may be overkill.

Instead, you can use a simple scripting language that is directly executable by a web browser. Two major web-oriented scripting languages are JavaScript (by Sun Microsystems) and VBScript (by Microsoft).

JavaScript is not full Java; it is a distinct scripting language based (loosely) on Java. Similarly, VBScript isn't full Visual Basic, its a language based on it.

Let's look at a simple JavaScript script that counts the Wednesdays in a month (if that's of interest to you!)

```
<SCRIPT LANGUAGE="JavaScript">
<!-- Begin
var now = new Date();
var month = now.getMonth();
var date = now.getDate();
var day = now.getDay();
var year = now.getFullYear();
m = new Array("January", "February",
"March", "April", "May", "June", "July",
"August", "September", "October",
"November", "December");
d = new Array("Sunday", "Monday",
"Tuesday", "Wednesday", "Thursday",
"Friday", "Saturday");
if (year < 2000) year = year + 1900;
//End of Month Calculations
var monarr = new Array(31, 28, 31,
30, 31, 30, 31, 31, 30, 31, 30, 31);
// check for leap year
if (((year % 4 == 0) && (year % 100
!= 0)) || (year % 400 == 0))
monarr[1] = "29";
```



```

//Finds the First Day for this Month
while (date != 1){
date = date - 1;
day = day - 1;
if (day < 0) day = day + 7;
}
//Count # of Wed in this month
weekday = 0;
while (date != monarr[month]){
date = date + 1;
day = day + 1;
if (day > 6)
day = 7 - day;
if (d[day] == "Wednesday")
weekday = weekday + 1;
}
//Fix For Months beginning on a Wed
if (d[day] == "Wednesday") weekday =
weekday + 1;
document.write (m[month] +
" contains " + weekday + " " + d[3] +
"s");// End -->
</script>

```

Like most scripting languages, in JavaScript there are no variable type declarations; types are dynamic. Control structures and expressions look like those found in Java. Though not shown, functions are included. The idea is to make it easy to add a bit of computation, while retaining most of the concepts and notation of conventional programming languages.

# Python

One of the newest and most innovative scripting languages is *Python*, developed by Guido van Rossum in the mid-90s. Python is named after the BBC “Monty Python” television series.

Python blends the expressive power and flexibility of earlier scripting languages with the power of object-oriented programming languages.

It offers a lot to programmers:

- An interactive development mode as well as an executable “batch” mode for completed programs.
- Very reasonable execution speed. Like Java, Python programs are compiled. Also like Java, the compiled code is in

an intermediate language for which an interpreter is written. Like Java this insulates Python from many of the vagaries of the actual machines on which it runs, giving it portability of an equivalent level to that of Java. Unlike Java, Python retains the interactivity for which interpreters are highly prized.

- Python programs require no compilation or linking. Nevertheless, the semi-compiled Python program still runs much faster than its traditionally interpreted rivals such as the shells, *awk* and *perl*.
- Python is freely available on almost all platforms and operating systems (Unix, Linux, Windows, MacOs, etc.)

- Python is completely *object oriented*. It comes with a full set of objected oriented features.
- Python presents a first class object model with first class functions and multiple inheritance. Also included are classes, modules, exceptions and late (run-time) binding.
- Python allows a clean and open program layout. Python code is less cluttered with the syntactic “noise” of declarations and scope definitions. Scope in a Python program is defined by the *indentation* of the code in question. Python thus breaks with current language designs in that white space has now once again acquired significance.

- Like Java, Python offers automated memory management through run-time reference counting and garbage collection of unreferenced objects.
- Python can be embedded in other products and programs as a control language.
- Python's interface is well exposed and is reasonably small and simple.
- Python's license is truly public. Python programs can be used or sold without copyright restrictions.
- Python is extendable. You can dynamically load compiled Python, Python source, or even dynamically load new machine (object) code to provide new features and new facilities.

- Python allows low-level access to its interpreter. It exposes its internal plumbing to a significant degree to allow programs to make use of the way the plumbing works.
- Python has a rich set of external library services available. This includes, network services, a GUI API (based on tcl/Tk), Web support for the generation of HTML and the CGI interfaces, direct access to databases, etc.

# Using Python

Python may be used in either interactive or batch mode.

In interactive mode you start up the Python interpreter and enter executable statements. Just naming a variable (a trivial expression) evaluates it and echoes its value.

For example (>>> is the Python interactive prompt):

```
>>> 1
1
>>> a=1
>>> a
1
>>> b=2.5
>>> b
2.5
```



```
>>> a+b
3.5
>>> print a+b
3.5
```

You can also incorporate Python statements into a file and execute them in batch mode. One way to do this is to enter the command

```
python file.py
```

where **file.py** contains the Python code you want executed. Be careful though; in batch mode you must use a **print** (or some other output statement) to force output to be printed. Thus

```
1
a=1
a
```

```
b=2.5
```

```
b
```

```
a+b
```

```
print a+b
```

when run in batch mode prints only 3.5 (the output of the `print` statement).

You can also run Python programs as Unix shell scripts by adding the line

```
#!/usr/bin/env python
```

to the head of your Python file.

(Since `#` begins Python comments, you can also feed the same augmented file directly to the Python interpreter)

# Python Command Format

In Python, individual primitive commands and expressions must appear on a single line.

This means that

```
a = 1
+b
```

does not assign `1+b` to `a`! Rather, it assigns `1` to `a`, then evaluates `+b`.

If you wish to span more than one line, you must use `\` to escape the line:

```
a = 1      \
+b
```

is equivalent to

```
a = 1 +b
```

Compound statements, like `if` statements and `while` loops, *can* span multiple lines, but individual statements within an `if` or `while` (if they are primitive) must appear one a single line.

Why this restriction?

With it, `;`'s are mostly unnecessary!

A `;` at the end of a statement is legal but usually unnecessary, as the end-of-line forces the statement to end.

You can use a `;` to squeeze more than one statement onto a line, if you wish:

```
a=1; b=2 ; c=3
```

# Identifiers and Reserved Words

Identifiers look much the same as in most programming languages. They are composed of letters, digits and underscores. Identifiers must begin with a letter or underscore. Case is significant. As in C and C++, identifiers that begin with an underscore often have special meaning.

Python contains a fairly typical set of reserved words:

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

# Numeric Types

There are four numeric types:

1. Integers, represented as a 32 bit (or longer) quantity. Digits sequences (possibly) signed are integer literals:

**1      -123    +456**

2. Long integers, of unlimited precision. An integer literal followed by an **L** or **l** is a long integer literal:

**123456789000000000000000L**

3. Floating point values, represented as a 64 bit floating point number. Literals are of fixed decimal or exponential form:

**123.456      1e10    6.0231023**

4. Complex numbers, represented as a pair of floating point numbers. In complex literals **j** or **J** is used to

denote the imaginary part of the complex value:

`1.0+2.0j`   `-22.1j`   `10e10J+20.0`

There is no character type. A literal like `'a'` or `"c"` denotes a string of length one.

There is no boolean type. A zero numeric value (any form), or `None` (the equivalent of void) or an empty string, list, tuple or dictionary is treated as false; other values are treated as true.

Hence

`"abc"` and `"def"`

is treated as true in an `if`, since both strings are non-empty.

# Arithmetic Operators

<u>Op</u>	<u>Description</u>
**	Exponentiation
-	Unary plus
+	Unary minus
~	Bit-wise complement (int or long only)
*	Multiplication
/	Division
%	Remainder
-	Binary plus
+	Binary minus
<<	Bit-wise left shift (int or long only)
>>	Bit-wise right shift (int or long only)
&	Bit-wise and (int or long only)
	Bit-wise or (int or long only)
^	Bit-wise Xor (int or long only)
<	Less than
>	Greater than



<b>&gt;=</b>	Greater than or equal
<b>&lt;=</b>	Less than or equal
<b>==</b>	Equal
<b>!=</b>	Not equal
<b>and</b>	Boolean and
<b>or</b>	Boolean or
<b>not</b>	Boolean not

# Operator Precedence Levels

Listed from lowest to highest:

<b>or</b>	Boolean OR
<b>and</b>	Boolean AND
<b>not</b>	Boolean NOT
<b>&lt;, &lt;=, &gt;, &gt;=, &lt;&gt;, !=, ==</b>	Comparisons
<b> </b>	Bitwise OR
<b>^</b>	Bitwise XOR
<b>&amp;</b>	Bitwise AND
<b>&lt;&lt;, &gt;&gt;</b>	Shifts
<b>+, -</b>	Addition and subtraction
<b>*, /, %</b>	Multiplication, division, remainder
<b>**</b>	Exponentiation
<b>+, -</b>	Positive, negative (unary)
<b>~</b>	Bitwise not

# Arithmetic Operator Use

Arithmetic operators may be used with any arithmetic type, with conversions automatically applied. Bit-wise operations are restricted to integers and long integers. The result type is determined by the “generality” of the operands. (Long is more general than int, float is more general than both int and long, complex is the most general numeric type). Thus

```
>>> 1+2
```

```
3
```

```
>>> 1+111L
```

```
112L
```

```
>>> 1+1.1
```

```
2.1
```

```
>>> 1+2.0j
```

```
(1+2j)
```

Unlike almost all other programming languages, relational operators may be “chained” (as in standard mathematics).

Therefore

**a > b > c**

means **(a > b) and (b > c)**

# Assignment Statements

In Python assignment is a statement *not* an expression.

Thus

```
a+(b=2)
```

is illegal.

Chained assignments are allowed:

```
a = b = 3
```

Since Python is dynamically typed, the type (and value) associated with an identifier can change because of an assignment:

```
>>> a = 0
```

```
>>> print a
```

```
0
```

```
>>> a = a + 0L
```

```
>>> print a
```

0L

```
>>> a = a + 0.0
```

```
>>> print a
```

0.0

```
>>> a = a + 0.0j
```

```
>>> print a
```

0j

# If and While Statements

Python contains `if` and `while` statements that are fairly similar to those found in C and Java.

There are some significant differences though.

A line that contains an `if`, `else` or `while` ends in a `:`. Thus we might write:

```
if a > 0:
    b = 1
```

Moreover the indentation of the then part is *significant*! You don't need `{` and `}` in Python because all statements indented at the same level are assumed to be part of the same block.

In the following Python statements

```
if a>0:
```

```
    b=1
```

```
    c=2
```

```
d=3
```

the assignments to **b** and **c** constitute then part; the assignment to **d** follows the if statement, and is independent of it. In interactive mode a blank line is needed to allow the interpreter to determine where the **if** statement ends; this blank line is not needed in batch mode.



# The if Statement

The full form of the `if` statement is

```
if expression:
    statement(s)
elif expression:
    statement(s)
...
else:
    statement(s)
```

Note those pesky `:`'s at the end of the `if`, `elif` and `else` lines. The expressions following the `if` and optional `elif` lines are evaluated until one evaluates to true. Then the following statement(s), delimited by indentation, are executed. If no expression evaluates to true, the statements following the `else` are executed.

Use of **else** and **elif** are optional; a “bare” **if** may be used.

If any of the lists of statements is to be null, use **pass** to indicate that nothing is to be done.

For example

```
if a>0:
    b=1
elif a < 0:
    pass
else
    b=0
```

This **if** sets **b** to 1 if **a** is > 0; it sets **b** to 0 if **a** == 0, and does nothing if **a** < 0.

# While Loops

Python contains a fairly conventional `while` loop:

```
while expression:  
    body
```

Note the ":" that ends the header line. Also, indentation delimits the body of the loop; no braces are needed. For example,

```
>>> a=0; b=0  
>>> while a < 5:  
...     b = b+a**2  
...     a= a+1  
...  
>>> print a,b  
5 30
```

# Break, Continue and Else in Loops

Like C, C++ and Java, Python allows use of **break** within a loop to force loop termination. For example,

```
>>> a=1
>>> while a < 10:
...     if a+a == a**2:
...         break
...     else:
...         a=a+1
...
>>> print a
2
```

A `continue` may be used to force the next loop iteration:

```
>>> a=1
>>> while a < 100:
...     a=a+1
...     if a%2==0:
...         continue
...     a=3*a
...
>>> print a
105
```

Python also allows you to add an **else** clause to a **while** (or **for**) loop.

The syntax is

```
while expression:
```

```
    body
```

```
else:
```

```
    statement(s)
```

The **else** statements are executed when the termination condition becomes false, but not when the loop is terminated with a **break**. As a result, you can readily program “search loops” that need to handle the special case of search failure:

```
>>> a=1
>>> while a < 1000:
...     if a**2 == 3*a-1:
...         print "winner: ",a
...         break
...     a=a+1
... else:
...     print "No match"
...
No match
```