

CSCI 553: Networking III

Unix Network Programming

Spring 2007



Basic Scripting



Today's Agenda

- Thursday's (2/14) Task
 - Untar & uncompress unp code
 - build, test basic datetime client
- Thursday (2/21) Task
 - Project version 0.1
 - definitive project.txt description
 - Add some design/features/issue discussions
 - A prototype or at least mock up of main function, some functionality
- Lab 03 Discussion
- Introduction to Scripting using Python



Lab 03

- Dependency tree

```
all : filter.o options.o main.o reverse.o palindrome.o  palindrome
```

```
palindrome : main.o filter.o options.o palindrome.o reverse.o
```

```
    gcc -o palindrome main.o filter.o options.o palindrome.o reverse.o
```

```
-----
```

```
palindrome: main.o filter.o options.o palindrome.o reverse.o
```

```
    gcc -c main.c filter.c options.c palindrome.c reverse.c
```

```
    gcc -o palindrome main.o filter.o options.o palindrome.o reverse.o
```

```
main.o: main.c
```

```
    gcc -c main.c
```



Lab 03

- Make Variables & Relative Directories

install:

```
cp -a libpalindrome.a /home/csci553/harry/harryrepo/lab03/lib  
cp -a *.h /home/csci553/harry/harryrepo/lab03/include
```



Lab 03

- **Make Variables & Relative Directories**

```
OUT_DIR=/home/csci553/harry/harryrepo/lab03
```

```
install:
```

```
cp -a libpalindrome.a $(OUT_DIR)/lib
```

```
cp -a *.h $(OUT_DIR)/include
```



Lab 03

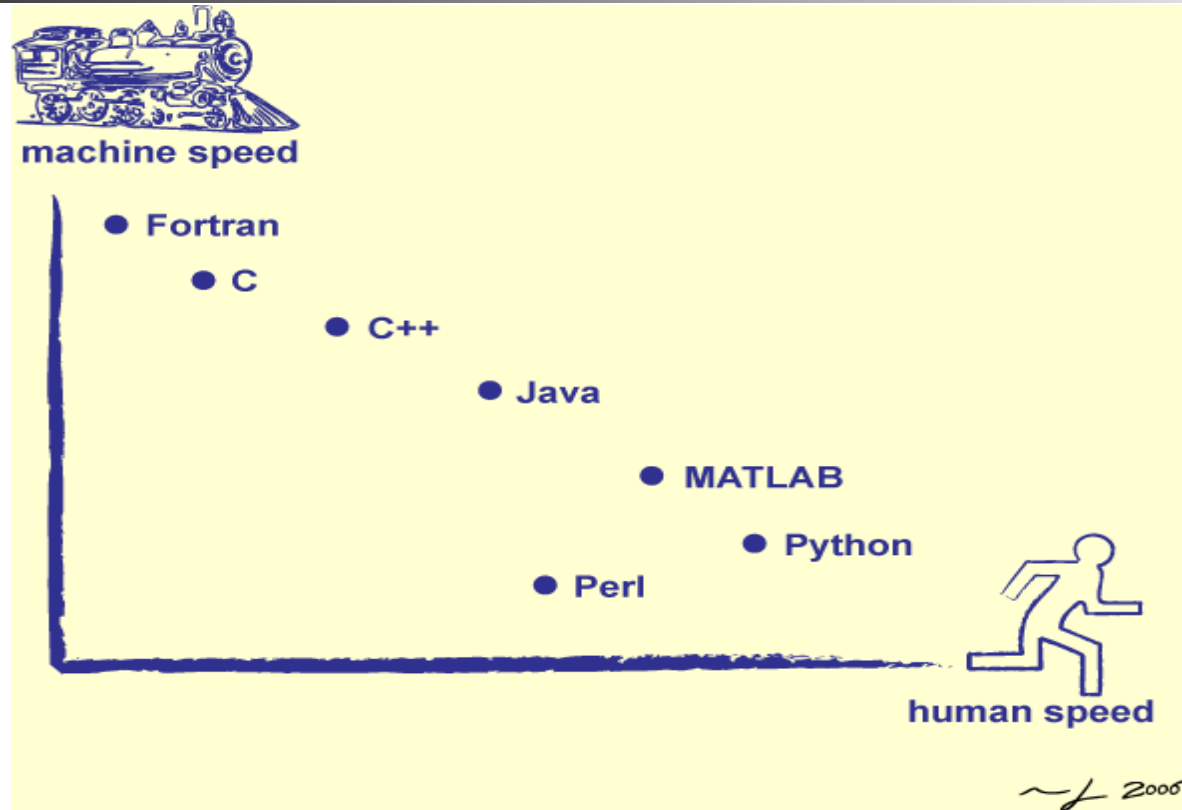
- Miscellaneous & other observations
 - Unix is case sensitive
 - Makefile makefile makeFile MakeFile
 - Oh and default for make is 1) makefile 2) Makefile
 - Try and understand what you are doing
 - subversion ci command always requires a log message
 - It takes time, experience & perserverence to get over the learning curve



Introduction

- Two things determine time to solution:
 - How long it takes to write a program (human time)
 - How long it takes that program to run (machine time)
- Different languages make different tradeoffs between these [**Prechelt 2000**]
 - High-level languages let programmers express their thoughts more quickly...
 - ...but the more abstract the language, the more slowly it runs

Introduction



- This series of lectures introduces a versatile high-level language called **Python**
 - A good way to build tools and crunch data



Python's Strengths

- As flexible as the shell, but with real data structures
 - Don't have to squeeze everything into lists of strings
- Freely available for many platforms
- Widely used and well documented
- Much easier to learn and read than Perl
 - Material that took three days to teach in Perl took only two to teach in Python
 - Follow-up surveys showed significantly higher retention rates



Python's Weaknesses

- *Nimble languages* like Python are slower than *sturdy languages* like Fortran, C/C++, or Java
 - Factor of 20 is common
 - But it's relatively easy to call libraries in those other languages from Python
- Doesn't have as many numerical or statistical tools as MATLAB
 - But its Numeric package isn't bad
- Not as widely used as Perl
 - But it's catching up



Why Another Language

- This course isn't really about Python
 - It's about solving software engineering problems
- But we have to write the examples in something
 - Might as well choose something useful
 - And it puts everyone on an equal footing
- For more information:
 - [[Lutz & Ascher 2003](#)] is the standard introduction
 - [[Martelli 2005](#)] is a collection of useful tips and tricks
 - See [Python Cookbook](#) for the on-line version



Execution Cycle

- **Sturdy languages** use a two-step execution cycle
 - Compile source code, putting machine-oriented form in file
 - Run the contents of that file on top of an operating system or *virtual machine*
- **Nimble languages** combine these two steps
 - Compiler and virtual machine are the same program
 - Load source code, translate into more compact form if necessary, and execute

Execution Cycle

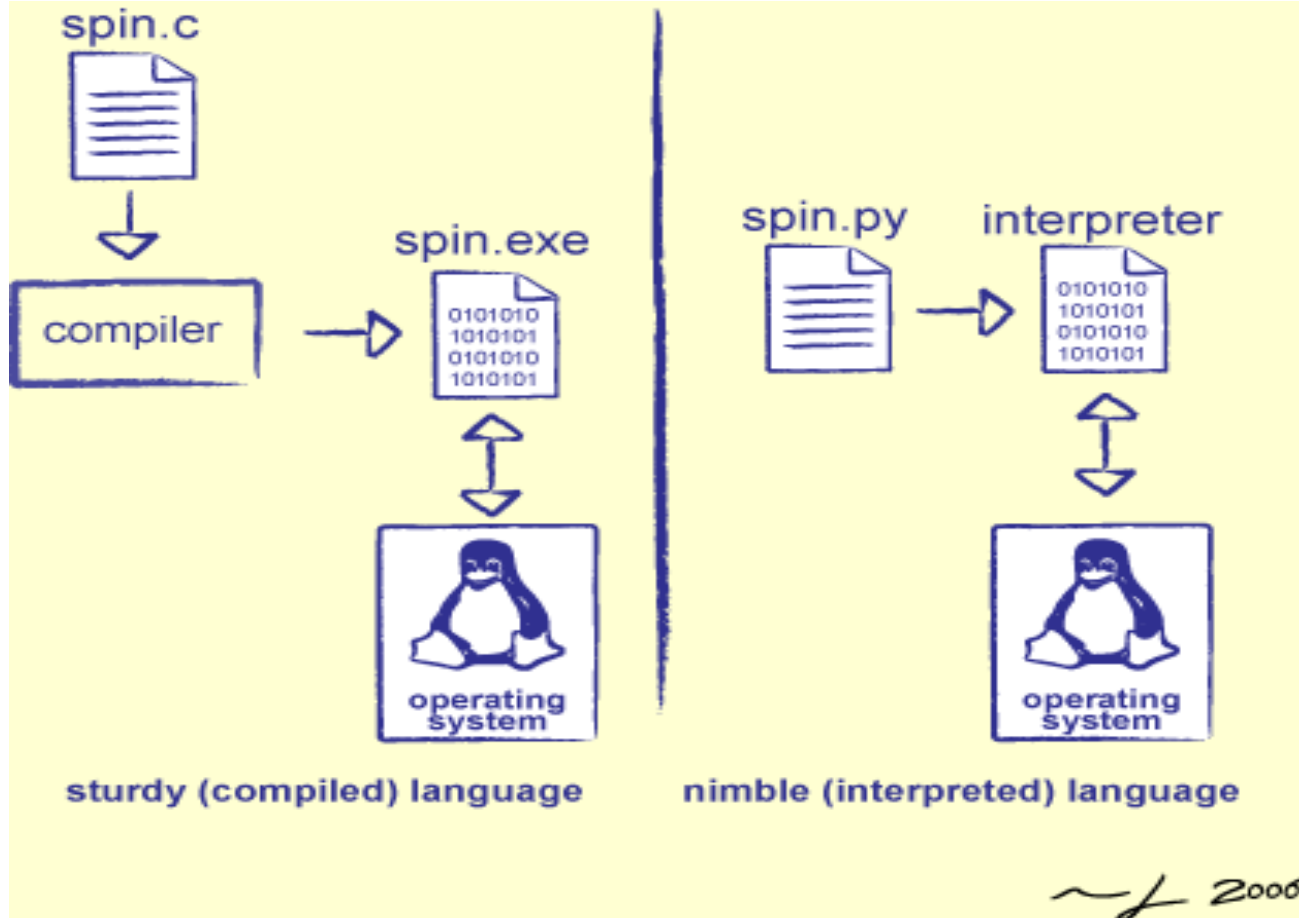


Fig 6.2: Sturdy vs. Nimble Execution



Execution Cycle

- This is why sturdy programs run faster, but nimble programs are faster to write
 - Compiling gives the computer a chance to optimize
 - Load-and-go makes human turnaround faster
 - And as we'll see **later**, it permits some powerful high-level programming tricks



Running Python Programs

- Interactively, like the shell

```
$ python
```

```
Python 2.4.2 (#67, Sep 28 2005, 12:41:11) [MSC v.1310 32 bit (Intel)] on win32  
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>> print 124/28
```

```
4
```

```
>>> print 124.0/28.0
```

```
4.4285714285714288
```

```
>>> ^D
```

- "^D" represents control-D, which is Unix's way of saying "end of input"

- Obviously don't have to retype program every time you want to run it

- Save program in a file with a .py extension, and type `python filename.py`

```
$ cat saved.py
```

```
print 124/28
```

```
print 124.0/28.0
```

```
$ python saved.py
```

```
4
```

```
4.42857142857
```



Execution Shortcuts

- On Unix, make `#!/usr/bin/python` the first line of the program
 - This tells Unix to execute `/usr/bin/python` with the rest of the file as its input
- Of course, this doesn't work if Python is installed somewhere else
 - Use `which python` to find out
- Better: use `#!/usr/bin/env python` as the first line
 - This tells Unix to use `/usr/bin/env` to find Python, then run the script with it

```
$ cat hashbang.py
#!/usr/bin/env python
print 124/28
print 124.0/28.0
$ hashbang.py
4
4.42857142857
```

- On Windows, associate `.py` files with Python
 - Happens automatically when you run the Python Windows installer
 - Double-clicking on anything ending in `.py` will then run it

Variables

- Variables are names for values
 - No declarations: variables are created when something is assigned to them
- ```
planet = "Pluto"
moon = "Charon"
p = planet
```

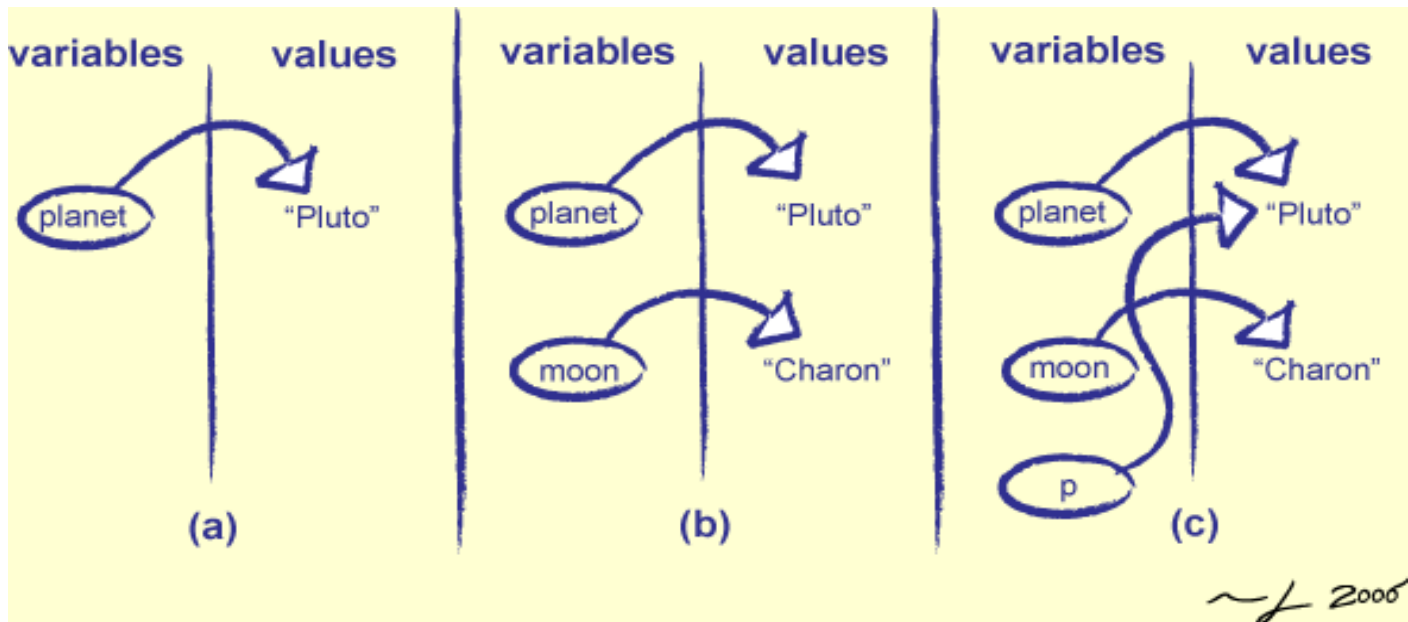


Fig 6.3: Variables Refer to Values

# Variables

- No types: a variable is just a name, and can refer to different types of values at different times

```
planet = "Pluto"
moon = "Charon"
p = planet
planet = 9
```

- Your code will be easier to understand if you don't abuse this

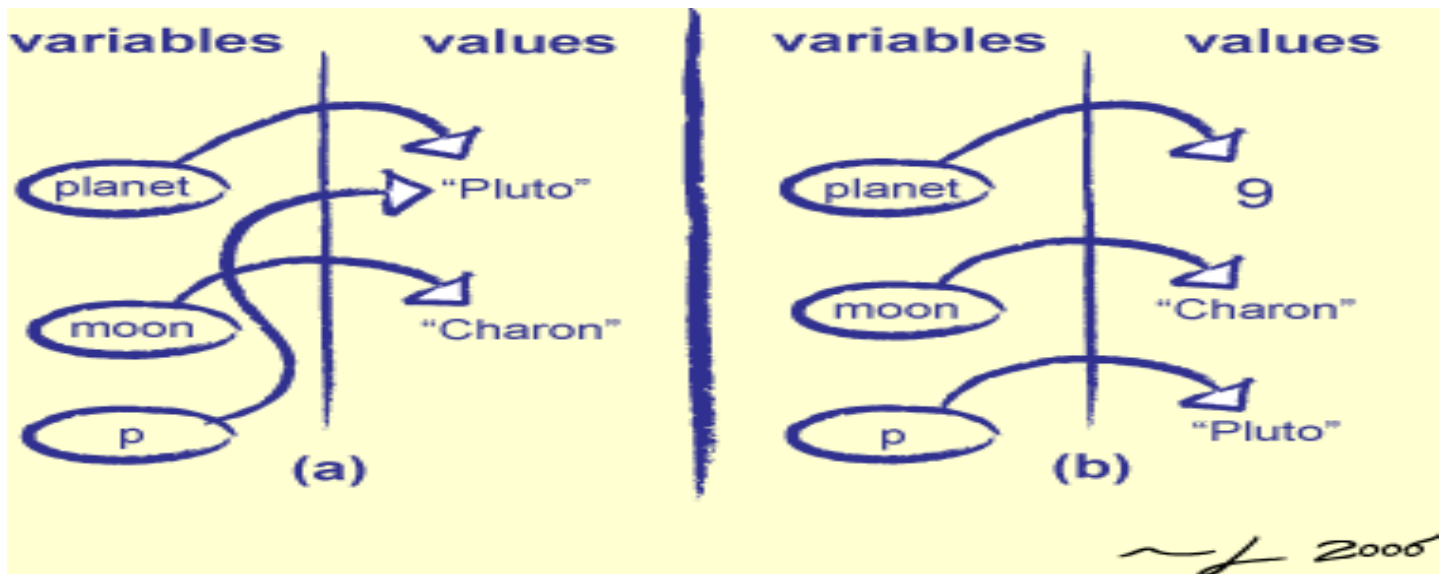


Fig 6.4: Variables Are Untyped



# Possible Mistakes

---

- Must give a variable a value before using it

```
planet = "Sedna"
```

```
print plant # note the misspelling
```

```
Traceback (most recent call last):
```

```
 File "lec/inc/py01/undefined_var.py", line 2, in ?
```

```
 print plant # note the misspelling
```

```
NameError: name 'plant' is not defined
```

- Unlike some languages, Python doesn't provide a default value
    - Because doing so can hide a lot of errors
  - Note: anything from "#" to the end of the line is a comment
- 
- Variables don't have types, but values do

- Python complains if you try to operate on incompatible values

```
x = "two" # "two" is a string
```

```
y = 2 # 2 is an integer
```

```
print x * y # multiplying a string concatenates it repeatedly
```

```
twotwo
```

```
print x + y # but you can't add an integer and a string
```

```
Traceback (most recent call last):
```

```
 File "lec/inc/py01/add_int_str.py", line 4, in ?
```

```
 print x + y # but you can't add an integer and a string
```

```
TypeError: cannot concatenate 'str' and 'int' objects
```



# Printing

---

- The print statement prints zero or more values to standard output
  - Separated by spaces
- Automatically puts a newline at the end
  - So print on its own just prints a blank line
  - Putting a comma at the end of the line suppresses the newline

```
planet = "Pluto"
num_moons = 1
moon = "Charon"
print planet, "has", num_moons, "satellite",
print "and its name is", moon
Pluto has 1 satellite and its name is Charon
```



# Quoting

---

- Use either single or double quotes to create strings
  - Each string must start and end with the same kind of quote
  - But different strings in the same program can use different kinds of quotes

```
print "He said, \"It ain't what you know, it's what you
can.\""
```

```
He said, "It ain't what you know, it's what you can."
```

- Use triple quotes (of either kind) to create a multi-line string

```
print "Sedna was discovered in 2004"
```

```
print 'It takes 10,500 years to circle the sun.'
```

```
print '''The tiny world may be part of the Oort Cloud,
a shell of icy proto-comets left over from
the formation of the Solar System.'''
```



# Converting Values to Strings

---

- The built-in function `str` converts things to strings

```
print "Diameter: " + str(1280) + "-" + str(1760) + " km"
```

```
Diameter: 1280-1760 km
```

- Use `int`, `float`, etc. to convert values to other types

```
print int(12.3)
```

```
print float(4)
```

```
12
```

```
4.0
```



# Escape Sequences

---

- Use *escape sequences* to put special characters in strings
  - Borrowed directly from C
  - Most common are tab `"\t"` and newline `"\n"`

| <b>Expression</b> | <b>Meaning</b>  |
|-------------------|-----------------|
| <code>\\</code>   | backslash       |
| <code>\'</code>   | single quote    |
| <code>\"</code>   | double quote    |
| <code>\b</code>   | backspace       |
| <code>\n</code>   | newline         |
| <code>\r</code>   | carriage return |
| <code>\t</code>   | tab             |



# Numbers

---

- 14 is an integer (32 bits long on most machines)
- 14.0 is double-precision floating point (64 bits long)
- $1+4j$  is a complex number (two 64-bit values)
  - Use `x.real` and `x.imag` to get the real and imaginary parts
- 123456789L is a *long integer*
  - Arbitrary length: uses as much memory as it needs to
  - Operations are several times slower



# Arithmetic

- Python borrows C's numeric operators
  - And adds `**` for exponentiation

| Name           | Operator | Use          | Value              | Notes                                              |
|----------------|----------|--------------|--------------------|----------------------------------------------------|
| Addition       | +        | 35 + 22      | 57                 |                                                    |
|                |          | 'Py + 'thon' | 'Python'           |                                                    |
| Subtraction    | -        | 35 - 22      | 13                 |                                                    |
| Multiplication | *        | 3 * 2        | 6                  |                                                    |
|                |          | 'Py' * 2     | 'PyPy'             | 2 * 'Py' is illegal                                |
| Division       | /        | 3.0 / 2      | 1.5                |                                                    |
|                |          | 3 / 2        | 1                  | Integer division rounds down: -3 / 2 is -2, not -1 |
| Exponentiation | **       | 2 ** 0.5     | 1.4142135623730951 |                                                    |
| Remainder      | %        | 13 % 5       | 3                  |                                                    |

- Python also has C's *in-place operators*
  - `x += 3` does the same thing as `x = x + 3`
  - `5 += 3` is an error, since you can't assign a new value to 5...



# Booleans

---

- True and False are true and false (d'oh)
- Empty string and 0 are considered false
  - Just as 3 is equivalent to 3.0
- (Almost) everything else is true
- Combine Booleans using and, or, not

| Expression                 | Result     | Notes                                                                                  |
|----------------------------|------------|----------------------------------------------------------------------------------------|
| True or False              | True       |                                                                                        |
| True and False             | False      |                                                                                        |
| 'a' or 'b'                 | 'a'        | or is true if either side is true, so it stops after evaluating 'a'                    |
| 'a' and 'b'                | 'b'        | and is only true if both sides are true, so it doesn't stop until it has evaluated 'b' |
| 0 or 'b'                   | 'b'        | 0 is false, but 'b' is true                                                            |
| 0 and 'b'                  | 0          | Since 0 is false, Python can stop evaluating there                                     |
| 0 and (1/0)                | 0          | 1/0 would be an error, but Python never gets that far                                  |
| (x and 'set') or 'not set' | It depends | if x is true, this expression's value is 'set'; if x is false, it is 'not set'         |



# Short-Circuit Evaluation

---

- and and or are *short-circuit* operators
  - Evaluate expressions left to right
  - Stop as soon as answer is known
  - Result is the last thing evaluated (rather than True or False)
- Can be used to create expressions like `val = cond and left or right`
  - If `cond` is True, `val` is assigned left
  - If `cond` is False, `val` is assigned right
  - It works, but it's hard to read
  - Clever isn't always smart



# Comparisons

---

- Python borrows C's comparison operators, too
  - But allows you to chain comparisons together, just as in mathematics

| <b>Expression</b>             | <b>Value</b>                                          |
|-------------------------------|-------------------------------------------------------|
| <code>3 &lt; 5</code>         | True                                                  |
| <code>3.0 &lt; 5</code>       | True                                                  |
| <code>3 != 5</code>           | True                                                  |
| <code>3 == 5</code>           | False                                                 |
| <code>3 &lt; 5 &lt;= 7</code> | True                                                  |
| <code>3 &lt; 5 &gt;= 2</code> | True (but please don't write this – its hard to read) |
| <code>3+2j &lt; 5</code>      | Error: can only use == and != on complex numbers      |

- Note the difference between assignment and testing for equality
  - Use a single equals sign = for assignment
  - Use a double equals sign == to test if two things have equal values



# String Comparisons

---

- Characters are encoded as numbers: digits come before uppercase letters, all of which come before lowercase letters
  - Punctuation is mixed in between, just to make matters difficult
- Strings are compared character by character from first to last until:
  - One character is less than another
  - One string runs out of characters

| <b>Expression</b>             | <b>Value</b> |
|-------------------------------|--------------|
| <code>'abc' &lt; 'def'</code> | True         |
| <code>'abc' &lt; 'Abc'</code> | False        |
| <code>'ab' &lt; 'abc'</code>  | True         |
| <code>'0' &lt; '9'</code>     | True         |
| <code>'100' &lt; '2'</code>   | True         |



# Conditionals

---

- Python uses `if`, `elif` (*not* else if), and `else`
- Use a colon and indentation to show nesting

```
a = 3
```

```
if a < 0:
```

```
 print 'less'
```

```
elif a == 0:
```

```
 print 'equal'
```

```
else:
```

```
 print 'greater' greater
```



# Why Indentation?

---

- Why doesn't Python use begin/end or {...}?
  - Because studies showed that indentation is what everyone actually pays attention to
  - Just count the number of warnings in C/Java books about misleading indentation
- Doesn't matter how much you use, but:
  - Everything in the block must be indented the same amount
  - And most people use four spaces
- Please do *not* use tabs
  - Python interprets them as (up to) eight characters
  - But different editors may display them differently



# While Loops

---

Do something repeatedly as long as some condition is true

- Again, use colon and indentation to show nesting

```
num_moons = 3
while num_moons > 0:
 print num_moons
 num_moons -= 1
3
2
1
```

- Do the “something” zero times if the condition is false the first time it is tested

```
print 'before'
num_moons = -1
while num_moons > 0:
 print num_moons
 num_moons -=1
print 'after'
before
after
```

- If the condition is always true, the loop never ends

```
num_moons = 3
while num_moons > 0:
 print num_moons
oops --- forgot to subtract one
3
3
3
:
```



# Break and Continue

---

- Can break out of the middle of a loop using break

```
num_moons = 3
while True: # Looks like an infinite loop...
 print num_moons
 num_moons -= 1
 if num_moons <= 1:
 break # ...but there's a way out
3
2
```

- Can skip to the next iteration using continue

```
num_moons = 5
while num_moons > 0:
 print 'top:', num_moons
 num_moons -= 1
 if (num_moons % 2) == 0:
 continue
 print '...bottom:', num_moons
top: 5
top: 4
...bottom: 3
top: 3
top: 2
...bottom: 1
top: 1
```

- Don't abuse these

- A single test at the top of the loop makes code much easier to read



# String Formatting

---

- Python's % operator formats output
  - Left side is a string specifying how things are to be formatted
  - Right side is the values to be formatted
    - One value on its own...
    - ...or several values in parentheses, separated by commas
- `'here %s go' % 'we'` creates "here we go"
  - The "%s" in the left string means "insert a string here"
  - Creates a new string (since strings are immutable)



# Format Specifiers

---

- `'left %d right %d' % (-1, 1)` creates `"left -1 right 1"`
  - `"%d"` stands for "decimal integer"
- `'%04d' % 13` creates `"0013"`
  - The leading zero means "pad with zeroes"
  - The 4 means "at least 4 characters wide"
  - `'[%-4d]' % 13` creates `"[13 ]"`
- `'%6.4f %%' % 37.2` creates `"37.2000 %"`
  - A floating point number, at least six characters wide, padded on the left with spaces, with four characters after the decimal point
  - `"%%"` is translated into a single `"%"`
    - Just as `"\\"` is how you represent a single `"\"` in a string



# Supported Formats

---

| Format | Meaning                                   | Example                           | Output            |
|--------|-------------------------------------------|-----------------------------------|-------------------|
| "d"    | Signed decimal integer                    | '%d %d' % (13, 15)                | "13 15"           |
| "o"    | Unsigned octal (base-8)                   | '%o %o' % (13, 15)                | "15 17"           |
| "x"    | Lower case unsigned hexadecimal (base-16) | '%x %x' % (13, 15)                | "d f"             |
| "X"    | Upper case unsigned hexadecimal (base-16) | '%X %X' % (13, 15)                | "D F"             |
| "e"    | Lower case exponential floating point     | '%e' % 123.45                     | "1.234500e+02"    |
| "E"    | Upper case exponential floating point     | '%E' % 123.45                     | "1.234500E+02"    |
| "f"    | Decimal floating point                    | '%f' % 123.45                     | "123.4500"        |
| "s"    | String (converts other types using str()) | '%s %s %s' % ('nickel', 28.58.68) | "nickel 28 58.69" |



# Summary

---

- Scripting languages are increasingly popular because they optimize human time
  - Which is now more expensive than machine time in all but a handful of cases
- Python is one of the cleanest scripting languages around
  - A good tool in its own right
  - An excellent way to build other tools