# Scripting Part 2 (part 1 was shell): Introduction Python



## Argh! Why another language?

- It's efficient.
- It's easy to learn.
- It's quick to write in.
- It provides immediate feedback to the programmer.
- …
- Oh, and it's a bit of a problem to debug really big programs… but we'll ignore that for now.



## Python

- Guido van Rossum created Python in the early 90s
  - Named after *Monty **Python**'s Flying Circus*
- Python strengths
  - Simple, clean syntax
  - Elegant object-orientation
  - Good documentation
- Python is powerful
  - Efficient high-level data structures are part of the language
  - It has a very comprehensive set of standard libraries
  - It is not hard to implement new functions in C or C++



## Pythonic style

- Like most languages, Python has a preferred *idiom* for many operations.
  - "Pythonic" way of doing things.
  - for (;;) { } ... is a C idiom for an infinite loop
  - for (i=0;i<N;i++) ... is a C idiom for a N-cycle loop
- See:The Zen of Python (PEP 20)
  - Google "Python pep 20"
  - http://www.python.org/dev/peps/pep-0020/

## Complications and versions

- Python is young and still evolving.
- Two distinct versions in common use:
  - Python 2
  - Python 3
- They are the "same" language, but there are important differences.
- We will focus on the "classic" Python 2, ideally version 2.7 (the latest).



## Origin of Scripting Languages

- Scripting languages originated as *job control languages*
  - 1960s: IBM System 360 had the Job Control Language
  - *Scripts* used to control other programs
    - » Launch compilation, execution
    - » Check return codes
- Scripting languages got increasingly more powerful in the UNIX world
  - Shell programming (which we've seen) was the start
  - Also AWK, Tcl/Tk, Perl
  - *Scripts* used to combine *components*
    - » *Gluing* applications [Ousterhout, 97 (see class web page)]



## "System Programming" Languages

- System programming languages (eg. C) replaced assembly languages (e.g. CS 273)
  - Benefits:
    - » The compiler hides unnecessary details, so these languages have a higher level of abstraction, increasing productivity
    - » They are *strongly typed*, *i.e.* meaning of information is specified before its use, enabling substantial error checking at compile time
    - » They make programs portable (if written correctly)
      - » JAVA attempts to ensure that they are portable by default
  - Both intended to write application from scratch
  - System programming langauges tried to minimize the loss in performance with respect to assembly languages
  - *E.g.* PL/1, Pascal, C, C++, Java



## Higher-level Programming

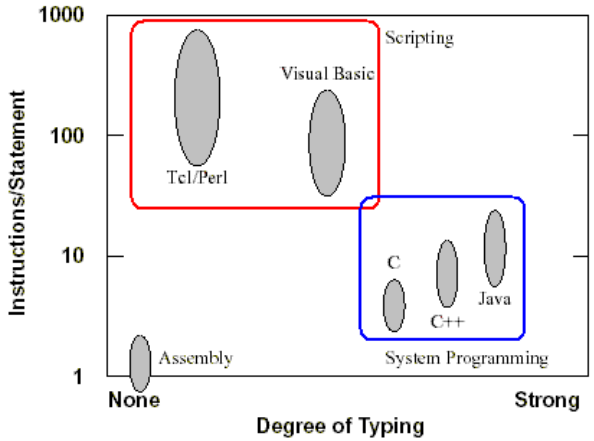
- Scripting languages provide an even higher-level of abstraction
  - The main goal is programming **productivity**
    - » Performance is a secondary consideration
  - Modern SL provide primitive operations with greater functionality
- Scripting languages are often interpreted, not compiled
  - Interpretation increases speed of development
    - » Immediate feedback
  - Compilation to an intermediate format is common

# Script Programming

- They are *dynamically* or *weakly typed*
  - *I.e.* Meaning of information is inferred
  - ✖ Less error checking at compile-time
    - » Run-time error checking is less efficient, but possible
  - ✓ Weak typing increases speed of development
    - » More flexible interfacing
    - » Fewer lines of code
    - » More real-time debugging
- They are not usually appropriate for
  - Efficient/low-level programming
  - Large programs



# Typing and Productivity




[Ousterhout, 97]



# Preview

- It's easy to try... [ live demo ]



# Running python (demo)

```
6%
6%
6% date
Sun Mar 25 18:53:13 EDT 2012
7%
7%
7% python
Python 2.6.1 (r261:67515, Jun 24 2010, 21:47:49)
[GCC 4.2.1 (Apple Inc. build 5646)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> print "hello"
hello
>>>
>>> x = 1
>>> y = x+2
>>>
```

## Backup slide if live demo fails

```
% python
Python 2.6.5 (r265:79063, Apr 16 2010, 13:57:41)
>>> 2+2
4
>>> print 2+2
4
>>> if 2+2 > 3: print "Bigger"
...
Bigger
>>>
>>> import random
>>> random.random()
0.88903728993504993
>>>  [ CONTROL-D ]
%
```



Quickstart:

1. Use “print” to print!
   **print "The value of x is",x**
2. Variables do not have to be declared before use.
3. Sub-statements are denoted by a colon “:”
   **if x>1: print "x is bigger than one."**



## Built-in Data Structures: Numbers

- Integers, floating-point numbers, complex numbers, arbitrarily long integers
  - 345
  - 3.45
  - 3+45j
  - 59802738573890250873 45L
- Operators
  - +, –, *, /, **, %,...
  - `abs()`, `floor()`,...



## Logic

| Operator | Meaning | Example | Result |
|---|---|---|---|
| == | equals | 1 + 1 == 2 | True |
| != | does not equal | 3.2 != 2.5 | True |
| < | less than | 10 < 5 | False |
| > | greater than | 10 > 5 | True |
| <= | less than or equal to | 126 <= 100 | False |
| >= | greater than or equal to | 5.0 >= 5.0 | True |

| Operator | Example | Result |
|---|---|---|
| and | 9 != 6 and 2 < 3 | True |
| or | 2 == 3 or -1 < 5 | True |
| not | not 7 > 0 | False |

# Numbers

- The usual suspects
  - » 12, 3.14, 0xFF, 0377, (-1+2)*3/4**5, abs(x), 0<x<=5
- C-style shifting & masking
  - » 1<<16, x&0xff, x|1, ~x, x^y
- Integer division truncates :-(
  - » 1/2 -> 0 # 1./2. -> 0.5, float(1)/2 -> 0.5
  - » Will be fixed in the future
- Long (arbitrary precision), complex
  - » 2L**100 -> 1267650600228229401496703205376L
    - » In Python 2.2 and beyond, 2**100 does the same thing
  - » 1j**2 -> (-1+0j)



# Built-in Data Structures: Strings

- Quotes sequences of characters
  - s = "hello"
- **TWO kinds of quote mark (equivalent semantics)**
  - `'Comp 206\nMessing with python today.'`
  - `"Comp 144\nMessing with python today."`
    - » Note we also have an embedded newline.
  - `"Python's tricks"`
  - `'He said "jump", and I said "How high?"'`
- Strings we don't worry about termination.



# Methods

- Recall, **methods** are like functions associated with a data type (i.e. a class).
- Methods
  - `s = "Hello"`
  - `s.upper()  -> "HELLO" (s itself is unchanged)`
  - `s.lower()  -> "hello"`
- upper and lower are methods of the string class.



# Built-in Data Structures: Strings

- Positional operators
  - Index `string[i]`
  - Slice `string[i:j]`
  - Length `len(string)`
- Formatting (extended `printf` notation)
  - `"This is %s %.1f" % ("python", 2.7)`

```
foo= "This is %s %.1f" % ("python", 2.7)
print foo
This is python 2.7
foo=2
print foo
2
```

  - `name = 'python'`
    `ver = 2.7`

**%** is a string **operator**, like addition for numbers.

## List-like structures

- Lists are collections of items.
- 2 key kinds of collection:
  - The **array**, indexed by a number.
    - » Items in an array are intrinsically sequential, even though you can randomly access them.
  - The **dictionary**, indexed by a string
    - » Items in a hash are not intrinsically ordered
    - » The word "hash" comes from the Perl community.
    - » This kind of object is also known as a collection.
  - (a couple of other types exist too: sets, tuples [immutable lists] )



## Built-in Data Structures: Lists

- A list is an **ordered collection of objects**
- Lists can contain *any* type of objects, mixed together
- Lists are *mutable* (they can be changed).
- Examples

| | |
|---|---|
| `[]` | Empty list |
| `[1, "2", 3.0]` | Three-element list (3 types) |
| `[1, ["2", 4], 3.0]` | Nested list |



## Built-in Data Structures: Lists

- Ordered collection of objects (like an array)
  - They can contain *any* type of object
  - E.g.

| | |
|---|---|
| [] | Empty list |
| [1, "2", 3.0] | Three-element list |
| [1, ["2", 4], 3.0] | Nested list |

- Operators

| | |
|---|---|
| – Access | `list[index]` |
| – Deletion | `del list[index]` |
| – Length | `len(list)` |



## Reading Requirement

- *Python tutorial*
  - Read sections 1 & 3
  - http://docs.python.org/tutorial/
- You can download all the python documentation at
  - http://docs.python.org/download.html
    or else see
  - http://www.cim.mcgill.ca/~dudek/206.html

## Preview: Python uses *modules*

- import modules (libraries) to acquire functionality.
  - string - string handling
  - re - regular expressions
  - os, sys - system stuff (eg. stdio, system calls)
  - random - random numbers
- More on this, with details, later.
- Example
  - `import os`
  - `os.system("date")`
  - `os.environ`
  - `import sys`
  - `sys.stdin`



## Printing tricks

- Extended print:

```
print “hello”                    → prints on stdout
sys.stdout.write("hello")        → same
sys.stderr.write("hello")        → prints on stderr
print >>sys.stderr, "hello"      → same

outfile = open("logfile.txt", "w")
print >>outfile, "hello"         → prints to “logfile.txt”
```



## Built-in Data Structures: Lists

- Operators
  - Concatenation +
    - » `[1, 2] + [3, 4] + [5]`
  - Repetition *
    - » `foo = [1, 2] * 5`
- Positional operators
  - Index `list[i]`
  - Slice `list[i:j]`

```
bar= foo[2:5]
print bar
 [ 2, 1, 2, 1, 1 ]
```

  - Length `len(list)`
- Generation
  - Ranges `range(start,end,step)`



## Lists: Accessing Items

- Syntax: **`list[index]`**
  - Indexing from the left starts at 0, with bounds checking.
  - *E.g.*

```
>>> l = [1, ["2", 4], 3.0]
>>> l[0]
1
>>> l[2]
3.0
>>> l[1]
['2', 4]
>>> l[3] = 4
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in ?
    l[3] = 4
IndexError: list assignment index out of range
```

## Lists: Accessing Items

- Syntax: **`list[-index]`**
  - Aside: what does this do in C?



## Negative indices (in C)

```
char a[128];
char *b;

strcpy(a,"Hello! my name is Nelson.");
b = &(a[7]);
b[0]='M';
printf("1:%s\n",b);          1:My name is Nelson.
b[-2] = ',';
printf("2:%s\n",a);          2:Hello, My name is Nelson.
```



## Lists: Accessing Items

- Syntax: **`list[-index]`**
  - Indexing from the **right** denoted by minus. Rightmost is -1
  - *E.g.*

```
>>> l = [1, ["2", 4], 3.0]
>>> l[-1]
3.0
>>> l[-3]
1
>>> l[-4]
Traceback (most recent call last):
  File "<pyshell#29>", line 1, in ?
    l[-4]
IndexError: list index out of range
```



## Lists: Deleting Items

- Syntax: **`del list[index]`**
  - *E.g.*

```
>>> l = [1, ["2", 4], 3.0]
>>> del l[2]
>>> l
[1, ['2', 4]]
>>> del l[2]
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in ?
    del l[2]
IndexError: list assignment index out of range
```

## Lists: Length

- Syntax: `len(list)`
  - *E.g.*

```
>>> l = [1, ["2", 4], 3.0]
>>> len(l)
3
>>> l = []
>>> len(l)
0
```



## Lists: Constructing Lists

- Concatenation
  - Syntax: `list1 + list2`
  - *E.g.*

```
>>> l1 = [1, 2]
>>> l1 + [3, 4, 5]
[1, 2, 3, 4, 5]
```

- Repetition
  - Syntax: `list * integer`
  - E.g.

```
>>> [1, 2] * 5
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```



## Lists: Constructing Lists

- Slicing
  - Syntax: `list[i:j]`
  - *E.g.*

```
>>> l = [1, ["2", 4], 3.0]
>>> l[1:2]
[['2', 4]]
>>> l[0:-2]
[1]
>>> l[1:-2]
[]
>>> l[1:-3]
[]
>>> l[1:3] = [2, 3]
>>> l
[1, 2, 3]
```



## Lists: Constructing Lists

- Ranges
  - Syntax: `range(start, end, step)`
  - Default values for start (0) and step (1)
  - *E.g.*

```
>>> range(1,100,10)
[1, 11, 21, 31, 41, 51, 61, 71, 81, 91]
>>> range(1,13)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
>>> range(3)
[0, 1, 2]
```

## Lists: Methods

- Inserting an item at a given position
  - Syntax: `list.insert[index, item]`
  - *E.g.*

```
>>> l = [1, ["2", 4], 3.0]
>>> l.insert(0, 8.3)
>>> l
[8.3, 1, ['2', 4], 3.0]
```

- Adding an item at the end of the list
  - Syntax: `list.append(item)`
  - *E.g.*

```
>>> l.append("end")
>>> l
[8.3, 1, ['2', 4], 3.0, "end"]
```



## Lists: Methods

- Sorting
  - Syntax: `list.sort()`
  - *E.g.*

```
>>> l = [1, 3, 2.0, 4]
>>> l.sort()
>>> l
[1, 2.0, 3, 4]
>>> l=["c", "d", "a", "b"]
>>> l.sort()
>>> l
['a', 'b', 'c', 'd']
```



## Lists: Methods

- Reversing
  - Syntax: `list.reverse()`
  - *E.g.*

```
>>> l = [1, 3, 2.0, 4]
>>> l.reverse()
>>> l
[4, 2.0, 3, 1]
```



## Strings

| | | |
|---|---|---|
| » "hello"+"world" | "helloworld" | # concatenation |
| » "hello"*3 | "hellohellohello" | # repetition |
| » "hello"[0] | "h" | # indexing |
| » "hello"[-1] | "o" | # (from end) |
| » "hello"[1:4] | "ell" | # slicing |
| » len("hello") | 5 | # size |
| » "hello" < "jello" | 1 | # comparison |
| » "e" in "hello" | 1 | # search |

» "escapes: \n etc, \033 etc, \if etc"

» 'single quotes' """triple quotes""" r"raw strings"

## Lists

- Flexible arrays, *not* Lisp-like linked lists
  - » a = [99, "bottles of beer", ["on", "the", "wall"]]
- Same operators as for strings
  - » a+b, a*3, a[0], a[-1], a[1:], len(a)
- Item and slice assignment
  - » a[0] = 98
  - » a[1:2] = ["bottles", "of", "beer"]
    -> [98, "bottles", "of", "beer", ["on", "the", "wall"]]
  - » del a[-1] # -> [98, "bottles", "of", "beer"]



## More List Operations

```
>>> a = range(5)          # [0,1,2,3,4]
>>> a.append(5)           # [0,1,2,3,4,5]
>>> a.pop()                    # [0,1,2,3,4]
5
>>> a.insert(0, 42)            # [42,0,1,2,3,4]
>>> a.pop(0)                   # [0,1,2,3,4]
5.5
>>> a.reverse()           # [4,3,2,1,0]
>>> a.sort()                   # [0,1,2,3,4]
```



## Control structures

- Same ideas a C, different syntax
- *if*
- *for*
- *while*
- *return*
- *break, continue*
- but no *switch*



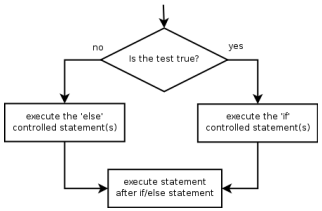
## IF

- If statement, much like C
- Sub-clause indicated by indentation!

```
if x>2:
  print "x is small"
  sizeflag=0
else:
  print "x was big"
  sizeflag=1
```

- **`if/else` statement**: Executes one block of statements if a certain condition is True, and a second block of statements if it is False.
  - Syntax:

```
if condition:
    statements
else:
    statements
```

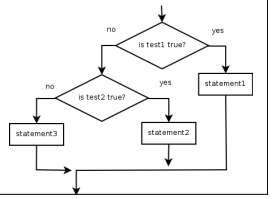



- Example:

```
gpa = 1.4
if gpa > 2.0:
    print "Welcome to Mars University!"
else:
    print "Your application is denied."
```

- Multiple conditions can be chained with `elif` ("else if"):

```
if condition:
    statements
elif condition:
    statements
else:
    statements
```






## Sequence Iteration (for)

- Syntax:

```
for var in sequence:
    statements
```

  - *E.g.*

```
sum = 0
for i in range(1,10,2):
   sum = sum + i

sum
25
```

- Membership operator: `in`



## Iteration (while)

- Syntax:

```
while test:
    statements
```

  - *E.g.*

```
sum = 0
i = 1
while i < 10:
   sum = sum + i
   i = i + 2

sum
25
```

- `break` and `continue` are also possible



## Functions

- Syntax:

```
def name(parameters):
        statements
        return object
```

  - *E.g.*

```
def incr(x):
    return x + 1

incr(3)
4
```

## parameters

- 2 kinds of parameters
  - non-keyword (i.e. positional) [familiar]
    ..., 3 ,...
  - keyword
    …, a=3 ,...
- Can supply default values
- Can have variable numbers of arguments



## Functions

- Default values
  - *E.g.*

```
def ask_ok(prompt, retries=4, complaint='Yes or no!'):
        while 1:
            ok = raw_input(prompt)
            if ok in ['y', 'ye', 'yes']: return 1
            if ok in ['n', 'no', 'nop', 'nope']:
                      return 0
            retries = retries - 1
            if retries < 0:
                     raise IOError, 'refusenik user'
            print complaint
```



## Functions

- Parameter passing by position and by name
  - E.g.

```
def parrot(voltage, state='a stiff', action='voom',
                type='Norwegian Blue',age=6):
        print "-- This parrot age ",age," wouldn't", action,
        print "if you put", voltage, "Volts through it."
        print "-- Lovely plumage, the", type
        print "-- It's", state, "!"


>>> parrot(1000)
>>> parrot(action = 'squawk', voltage = 1000000)
>>> parrot('a thousand', state = 'pushing up the daisies')
>>> parrot('a million', 'bereft of life', 'jump')
>>> parrot(action = 'VOOOOOM')
```



## Built-in Data Structures: Dictionaries

- A dictionary is an **unordered collection of objects indexed by keys (known as a hash in perl)**
- A*ny* object can be a key
- *Any* object can be a item indexed by a key
- Dictionaries are *mutable* (can be changed)
- Examples

`{ }` Empty dictionary

`{'item':'tire','price':20.99}` Two-element dictionary

## Dictionaries: Accessing items

- Syntax: **`list[key]`**
  - *E.g.*

```
>>> d = {'item':'tire','price':20.99}
>>> d['price']
20.99
>>> d[item]
Traceback (most recent call last):
  File "<pyshell#88>", line 1, in ?
    d[item]
NameError: name 'item' is not defined
>>> str = 'item'
>>> d[str]
'tire'
```



## Dictionaries: Deleting items

- Syntax: **`del list[key]`**
  - *E.g.*

```
>>> d = {'item':'tire','price':20.99}
>>> del d['item']
>>> d
{'price': 20.989999999999998}
>>> del d['brand']
Traceback (most recent call last):
  File "<pyshell#95>", line 1, in ?
    del d['brand']
KeyError: brand
```



## Dictionaries: Length

- Syntax: **`len(list)`**
  - *E.g.*

```
>>> d = {'item':'tire','price':20.99}
>>> len(d)
2
```



## Dictionaries: Methods

- Membership
  - Syntax: **`list.has_key(key)`**
  - *E.g.*

```
>>> l = {'item':'tire','price':20.99}
>>> l.has_key('item')
1
>>> l.has_key('brand')
0
```

## Dictionaries: Methods

- List of keys for a dictionary
  - Syntax: `list.keys()`
  - *E.g.*

```
>>> l = {'item':'tire','price':20.99}
>>> l.keys()
['item', 'price']
```

- List of values
  - Syntax: `list.values()`
  - *E.g.*

```
>>> l.values()
['tire', 20.989999999999998]
```



## Dictionaries

- Hash tables, "associative arrays"
  - » d = {"duck": "eend", "water": "water"}
- Lookup:
  - » d["duck"] -> "eend"
  - » d["back"] # raises KeyError exception
- Delete, insert, overwrite:
  - » del d["water"] # {"duck": "eend", "back": "rug"}
  - » d["back"] = "rug" # {"duck": "eend", "back": "rug"}
  - » d["duck"] = "duik" # {"duck": "duik", "back": "rug"}



## More Dictionary Ops

- Keys, values, items:
  - » d.keys() -> ["duck", "back"]
  - » d.values() -> ["duik", "rug"]
  - » d.items() -> [("duck","duik"), ("back","rug")]
- Presence check:
  - » d.has_key("duck") -> 1; d.has_key("spam") -> 0
- Values of any type; keys almost any
  - » {"name":"Guido", "age":43, ("hello","world"):1, 42:"yes", "flag": ["red","white","blue"]}



## Dictionary Details

- Keys must be **immutable**:
  - numbers, strings, tuples of immutables
    - » these cannot be changed after creation
  - reason is *hashing* (fast lookup technique)
  - **not** lists or other dictionaries
    - » these types of objects can be changed "in place"
  - no restrictions on values
- Keys will be listed in **arbitrary order**
  - again, because of hashing

## Functions (revisited)

- Functions can also have an arbitrary number of parameters
  - » Passed as a dictionary or as list of *remaining* parameters

```
def sum_args(*numbers):
    return sum(numbers)
print sum_args(2,8,1)       # prints: 11
```

def sum_args(**numbers): print numbers

```
print sum_args(a=1,this="that",eee=2.71)
```



## Built-in Data Structures: Tuples

- A tuple is an **ordered collection of objects**
  - – much like an array**, list**
- Tuples can contain *any* type of object
- Tuples are *immutable*
  - – Cannot be changed; think **const** in C
- Examples

```
()                  Empty tuple
1,                  One-element tuple (!)
(1, "2", 3.0)       Three-element tuple
1, ("2", 3.0)       Nested tuple
```



## Built-in Data Structures: Tuples

- **Commas** are used to define tuples
  - – Parentheses around tuples are optional
  - – *E.g.*

```
>>> 1,('2',2.0)
(1, ('2', 2.0))

>>> (1,('2',2.0))
(1, ('2', 2.0))
```

  - – The one-element list requires a trailing comma

```
>>> 1,
(1,)
>>> (1)          ← This is not a tuple but a number
1
```



## Tuples: Accessing Items

- Syntax: **`tuple[index]`**
  - – *E.g.*

```
>>> t = (1, 2, (3, 4, 5))
>>> t[1]
2
>>> t[-1]
(3, 4, 5)
>>> t[-1][1]
4
>>> t[3]
Traceback (most recent call last):
  File "<pyshell#110>", line 1, in ?
    t[3]
IndexError: tuple index out of range
```

# Tuples: No Deletion

- No deletion!
  - Tuples are immutable (cannot be changed)
- Length:
  - Syntax: **`len(tuple)`**
  - *E.g.*

```
>>> t = (1,2,(3,4,5))
>>> len(t)
3
>>> len(t[1])
Traceback (most recent call last):
  File "<pyshell#117>", line 1, in ?
    len(t[1])
TypeError: len() of unsized object
>>> len(t[2])
3
```



# Tuples: Constructing Tuples

- Concatenation
  - Syntax: **`tuple1 + tuple2`**
  - *E.g.*

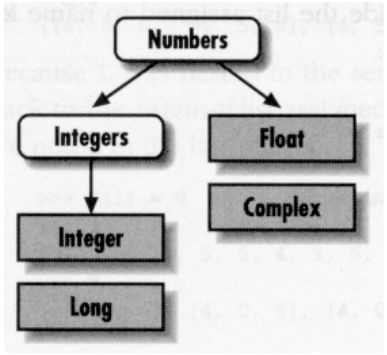
```
>>> t = (1,2) + (3,)
>>> t
(1, 2, 3)
```

- Repetition
  - Syntax: **`tuple * integer`**
  - E.g.

```
>>> t * 5
(1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3)
```



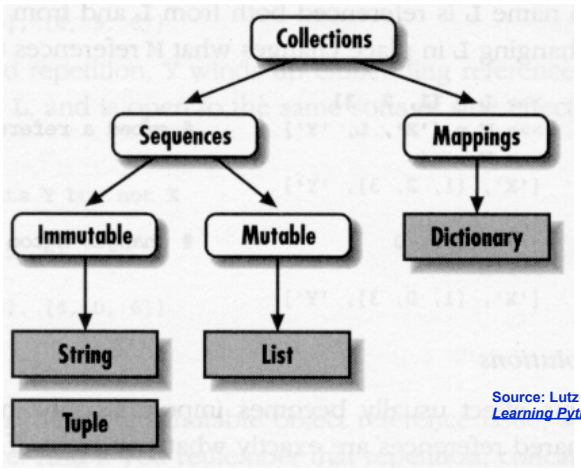
# Hierarchy of Numbers




Source: Lutz & Ascher, *Learning Python*, Figure 2-3



# Hierarchy of Built-in Collections




Source: Lutz & Ascher, *Learning Python*, Figure 2-3

## Statements: Assignment

- Syntax: **`reference = object`** or **`reference`**
  - *E.g.*

```
>>> a = 3
>>> a
3
>>> s1, n, m = "hello", 4.0, a
>>> s1
'hello'
>>> n
4.0
>>> m
3
```



## Variables

- No need to declare
- Need to assign (initialize)
  - » use of uninitialized variable raises exception
- Not typed

```
if friendly: greeting = "hello world"
else: greeting = 12**2
print greeting
```

»



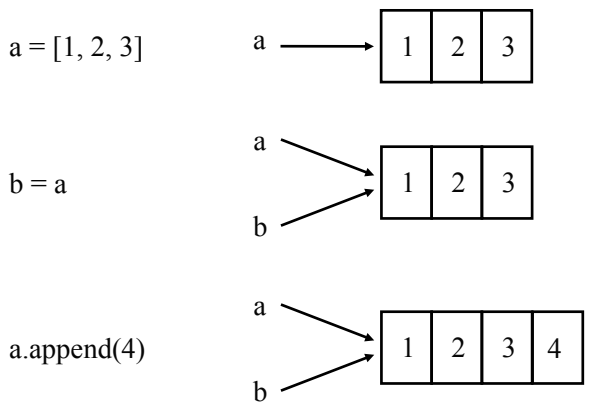
## Reference Semantics

- Assignment manipulates references
  - » x = y **does not make a copy** of y
  - » x = y makes x **reference** the object y references
- Very useful; but beware!
- Example:

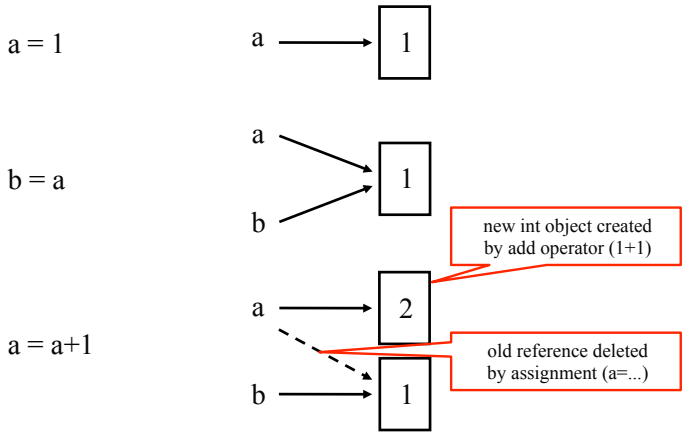
```
>>> a = [1, 2, 3]
>>> b = a
>>> a.append(4)
>>> print b
[1, 2, 3, 4]
```



## Changing a Shared List



## Changing an Integer






## Scope rules (subtle stuff)

- Python is unusual: if no global statement is in effect – assignments to names always go into the innermost scope.
- Recall: assignments do not copy data — they just bind names to objects.
- The same is true for deletions: the statement del x removes the binding of x from the namespace referenced by the local scope.



## Statements: Print (again)

- Syntax: **`print object`** or **`reference`**
  - *E.g.*

```
>>> print "hello", 'again'
hello again
>>> print 3.0e5
300000.0
>>> name = "python"
>>> ver = 2.2
>>> print "This is %(name)s %(ver).3f" % vars()
This is python 2.200
```



## Conditional (repeated)

- Syntax:

```
if test:
    statements
elif test:
    statements
else:
    statements
```

- Conditional expressions:

  `>, <, >=, <=, ==, and, or, not`

## Conditional

- *E.g.*

```
>>> x = -3
if x < 0:
         print "negative"
elif x == 0 or not x:
  print "zero-ish"
elif x >=7 and x<8:
  print "luck seven"
else:
  print "just plain positive"


negative
```



## Python Cont'd:

## Files, Modules, Classes, Exceptions and Examples



## Built-in functions

- Examples:
- **exec -- cool! Dynamic code execution.**
- **eval -- junior exec (expressions only)**
- **int, dict, list, float, complex**
- **ord, chr, ascii, bin, hex, str**
- **globals**
- **len**
- **map**
- **min, max, reversed, sum**
- **type**



## Knuth: an illustrious computer scientist

"We will perhaps eventually be writing only small modules which are identified by name as they are used to build larger ones, so that devices like indentation, rather than delimiters, might become feasible for expressing local structure in the source language."

- **Donald E. Knuth, "Structured Programming with goto Statements", Computing Surveys, Vol 6 No 4, Dec. 1974**

## Memory?

- How does allocation works?
- Variables are created when used. When are they disposed of?
  - Solution is *reference counting* and *garbage collection*.
  - When to variables contain references to a block of storage, it can be disposed of (like *free)*.
  - This means a *garbage collector* is being invoked (automatically) to check for is.



## Files

- Creating file object
  - Syntax: file_object = open(file name, mode)
    - » `input = open("inventory.dat", "r")`
    - » `output = open("report.dat", "w")`
- Manual close
  - Syntax: close(file_object)
    - » `close(input)`
- Reading an entire file
  - Syntax: string = file_object.read()
    - » `content = input.read()`
  - Syntax: list_of_strings = file_object.readlines()
    - » `lines = input.readlines()`



## Files

- Reading one line at time
  - Syntax: list_of_strings = file_object.readline()
    - » `line = input.readline()`
- Writing a string
  - Syntax: file_object.write(string)
    - » `output.write("Price is %(total)d" % vars()`
  - (Recall print works too)
- Writing a list of strings
  - Syntax: file_object.writelines(list_of_string)
    - » `output.writelines(price_list)`
- This is very simple!
  - Compare it with `java.io`



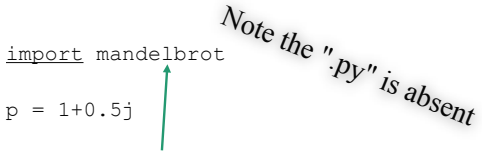
## Modules

- Example: **mandelbrot.py**

```
# Mandelbrot module
def inMandelbrotSet(point):
  """
  True iff point is in the Mandelbrot Set
  """
  X, t = 0 + 0j, 0
  while (t < 30):
          if abs(X) >= 2: return 0
          X, t = X ** 2 + point, t + 1
  return 1
```

## Using Modules

- Importing a module
  - Syntax: **import module_name**

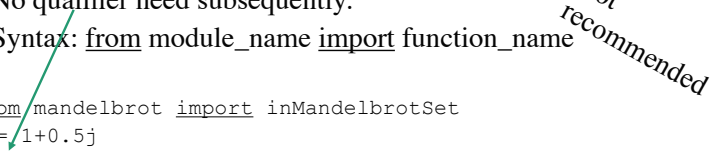



```
import mandelbrot

p = 1+0.5j

if mandelbrot.inMandelbrotSet(p):
        print "%f+%fj is in the set" % (p.real,
  p.imag)
else:
        print "%f+%fj is NOT in the set" % (p.real,
  p.imag)
```

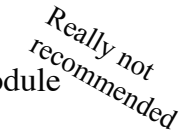


## Using Modules

- Importing individual functions within a module
  - No qualifier need subsequently.
  - Syntax: from module_name import function_name




```
from mandelbrot import inMandelbrotSet
p = 1+0.5j
if inMandelbrotSet(p):
        print "%f+%fj is in the set" % (p.real,
  p.imag)
else:
        print "%f+%fj is NOT in the set" % (p.real,
  p.imag)
```




- Importing all the functions within a module
  - Syntax: from module_name import *



## Standard Modules

- Python has a very comprehensive set of standard modules (a.k.a. libraries).
- It's one of the great strengths of the language.
  - See Python library reference
    - » http://www.python.org/doc/current/lib/lib.html



## random

- Reference
  - http://docs.python.org/library/random.html
- Some very useful functions
  - random.random()
  - random.choice(list)
    - » random.choice(['hello','howdy','hey'])
    - » random.choice(
  - random.gauss()
    - » number from normal (Gaussian) distribution.

## strings

- Standard strings have these without importing the module.
- Some very useful methods
  - find(s, sub[, start[,end]])
  - split(s[, sep[, maxsplit]])
  - strip(s)
  - **replace(str, old, new[, maxsplit])**
  - **lower(s)**
  - **count(s,w)**



## string example

**example1: lowercase all text**

```
for line in sys.stdin.readlines():
  line = string.lower(line)
  print line,
```

**example 2: count occurrences of cheese**

```
lines = file.readlines()
print string.count(string.join(lines),
 'cheese')
```



## Classes

- Defined using **`class`** and indentation
  - E.g.

```
class MyClass(parent):
        """A simple example class"""
        i = 12345
        def f(self):
            return 'hello world'
```

- *Methods* are functions defined within the class declaration or using the *dot* notation
- *Attributes* are variables defined within the the class declaration or using the *dot* notation



## Class Constructor

- `__init__` method
  - E.g.

```
class MyClass:
    def __init__(self):
                self.data = []
```

- Creation of instances is straightforward
  - E.g.

```
x = MyClass()
x.f()
```

Remember: an "instance" is an actual memory-using version of the generic "idea" represented by a class.

- Example

```
class Complex:
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart
        self.mymethod()
    def mymethod(self):
        return 0

x = Complex(3.0, -4.5)
>>> x.r, x.i
```



# Example Class

```
class Stack:
    "A well-known data structure…"
    def __init__(self):             # constructor
        self.items = []
    def push(self, x):
        self.items.append(x)        # the sky is the limit
    def pop(self):
        x = self.items[-1]          # what happens if it's empty?
        del self.items[-1]
        return x
    def empty(self):
        return len(self.items) == 0 # Boolean result
```



# Using Classes

- To create an instance, simply call the class object:

```
x = Stack()             # no 'new' operator!
```

- To use methods of the instance, call using dot notation:

```
x.empty() # -> 1
x.push(1)               # [1]
x.empty() # -> 0
x.push("hello")                 # [1, "hello"]
x.pop()         # -> "hello"    # [1]
```

- To inspect instance variables, use dot notation:

```
x.items         # -> [1]
```



# Subclassing

```
class FancyStack(Stack):
    "stack with added ability to inspect inferior stack items"

    def peek(self, n):
        "peek(0) returns top; peek(-1) returns item below that; etc."
        size = len(self.items)
        assert 0 <= n < size                    # test precondition
        return self.items[size-1-n]
```

## Subclassing (2)

```
class LimitedStack(FancyStack):
    "fancy stack with limit on stack size"

    def __init__(self, limit):
        self.limit = limit
        FancyStack.__init__(self)              # base class constructor

    def push(self, x):
        assert len(self.items) < self.limit
        FancyStack.push(self, x)               # "super" method call
```



## Class / Instance Variables

```
class Connection:
    verbose = 0                                # class variable
    def __init__(self, host):
        self.host = host                       # instance variable
    def debug(self, v):
        self.verbose = v                       # make instance variable!
    def connect(self):
        if self.verbose:                       # class or instance variable?
            print "connecting to", self.host
```



## Instance Variable Rules

- On use via instance (self.x), search order:
  - (1) instance, (2) class, (3) base classes
  - this also works for method lookup
- On assignment via instance (self.x = ...):
  - always makes an instance variable
- Class variables "default" for instance variables
- But...!
  - mutable *class* variable: one copy *shared* by all
  - mutable *instance* variable: each instance its own



## Scope: LEGB rules

- **L**. Local. (Names assigned in any way within a function (def or lambda), and not declared global in that function.
- **E**. Enclosing function locals. (Name in the local scope of any and all enclosing functions (def or lambda), from inner to outer.
- **G**. Global (module). Names assigned at the top-level of a module file, or declared global in a def within the file.
- **B**. Built-in (Python). Names preassigned in the built-in names module:open,range,...

## Scope Example

```
def f(x):
    global g
    y = 1
    print g,x,k,y

def w():
    print y

g=2
x=100
k=x
f("yes")
print g,x,k,y
y=444
f(1)
w()
```

**Creates *local* variable y**

**prints: 2 yes 100 1**

**would print: 2 100 100, but y is undefined so throws an error.**

**444**



## Other cool modules

- math - floor, exp, log, log10, pow, sin, gamma, pi,
- pickle - save & load stuctured data
- zlib, gzip - compression
- csv - process spreadsheet files
- time, os, sys
- subprocess
- posix - POSIX system calls
- urllib & cgi - web data & URL's
- http.server - web server



## Exceptions

- Exception handling ties 2 code blocks together:
  - If an exception (problem/error) occurs while executing the first block, then execute the second block instead.
- Typical execptions you might want to "catch"
  - undefined variables
  - illegal operations
  - bad math (underflow/overflow)
- You can also explicitly raise an exception based on your own criteria.



## Exceptions: try/except

- Syntax:

```
try:
    code block
except  [optional specific conditions]:
    code block
```

- Code blocks may contain functions and errors internal to them will be caught. Nesting is OK.

# Exceptions: a realistic example

- Requires that the value be numeric.
- Try/except/raise

```
while 1:
  try:
     x = int(raw_input("Please enter a number: "))
     break
  except ValueError:
     print "Oops! That was not valid.  Try again"
   except:  print "Horrible mytery error!!!"

print "Thank you."
print "Self-destruct will occur in ",x," seconds."
```



```
for i in moduleNames:
    if i == "Roombase": continue
    if i == "all": continue

     exec "import "+i          ->  import room19
     info = pyclbr.readmodule(i)
      for n in info.keys():   ->  n = room19
         try:
           mod=i+"."+n          -> r = room19.Room19()
           arr = "r="+mod+"()"
           exec arr
           arr = "r.arrive()"
           exec arr
```



# Example: random numbers

- **Generating a random arrangement of numbers between 1 and N, without replacement:**
- **import random**

```
numbers = range(1, N+1)
while numbers:
 j = random.choice(numbers)
 numbers.remove(j)
 print j
```



# Example: random lines

- **Randomizing the lines in a file:**
- import sys, random

```
lines = sys.stdin.readlines()
while lines:
        line = random.choice(lines)
        lines.remove(line)
        print line,
```

## random signature

```
import string, random
try:
   foo = open("/home/dudek/.sigdata").read()
   foo = string.split( foo, "\n\n" )
   map( string.strip, foo )
   for i in range(0,len(foo)):
      foo[i] = string.strip(foo[i])
   bar = random.choice(foo)
   foo2 = open('/home/keskoy/.signature','w')
   foo2.write(bar)
   foo2.close()
except:  print "boo hoo"
```



## Python: summary

- Is that everything?
  - Of course not
- We have seen a solid core of the language.
- Still missing
  - Scope rules:
    - » dynamic scope!
  - exec
  - multiple inheritance
  - linking with C or JAVA (easy)



## Python CGI programming

Common Gateway Interface: a protocol for passing data between a web server and a client script.



## A typical HTML form

Your first name:

Your last name:

Click here to submit form: Yeah!

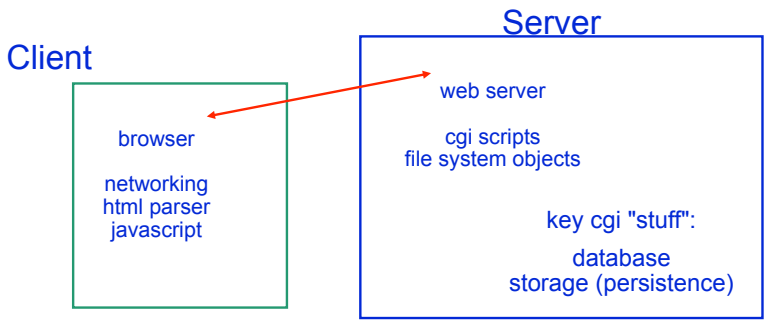
```
<form method="POST" action="http://host.com/cgi-bin/test.py">
   <p>Your first name: <input type="text" name="firstname">
   <p>Your last name: <input type="text" name="lastname">
   <p>Click here to submit form: <input type="submit" value="Yeah!">
   <input type="hidden" name="session" value="1f9a2">
</form>
```

## CGI Issues

- Web server passes information to CGI script
  - This data is encoded an in inconvenient to read.
  - Script passes back data to server which as returned to users as a web page they see in the browser.
  - The data going across the web most conform to the HTTP protocol
- Decoding data from CGI and re-encoding data to return can be accomplished easily in python via special libraries: **cgi** and **urllib** (and others).
  - Note, similar libraries exist for perl and C.



## Client-server connection (web)






## urllib

**urllib** provides file-like access to web pages

- urlopen(URL)
- read
- close

```
import urllib
print (urllib.urlopen("http://127.0.0.1/index.html")).read()
```



## SocketServer class

- 4 server classes
- TCPServer uses the Internet TCP protocol.
  - streams of data between the client and server.
  - Delivery and ordering guarantee
- UDPServer uses datagrams.
  - No ordering, no acknowledgement, unreliable

# SocketServer example

```
import SocketServer
class MyUDPHandler(SocketServer.BaseRequestHandler):
    """
    This class works similar to the TCP handler class, except that
    self.request consists of a pair of data and client socket, and since
    there is no connection the client address must be given explicitly
    when sending data back via sendto().
    """

    def handle(self):
        data = self.request[0].strip()
        socket = self.request[1]
        print "{} wrote:".format(self.client_address[0])
        print data
        socket.sendto(data.upper(), self.client_address)

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    server = SocketServer.UDPServer((HOST, PORT), MyUDPHandler)
    server.serve_forever()
```



# UDP client (Sends messages)

- CLIENT

```
import sys
import socket

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# SOCK_DGRAM is the socket type to use for UDP sockets
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# As you can see, there is no connect() call; UDP has no connections.
# Instead, data is directly sent to the recipient via sendto().
sock.sendto(data + "\n", (HOST, PORT))
received = sock.recv(1024)

print "Sent:     {}".format(data)
print "Received: {}".format(received)
```

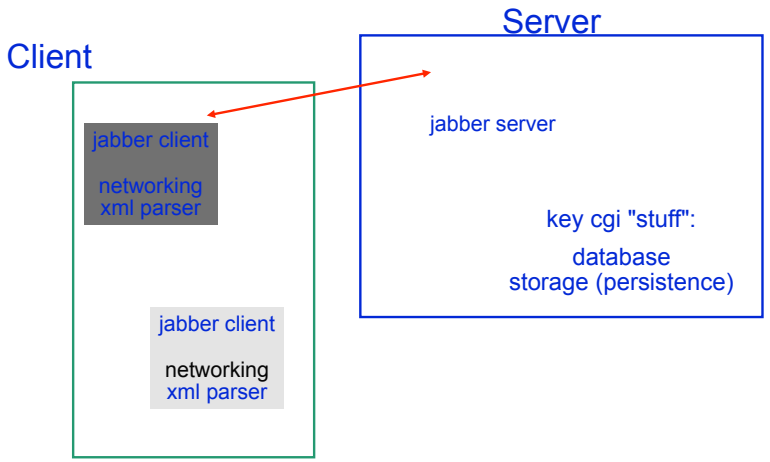


# Network Sockets

- Basic primitive for network communication.
  - Covered in detail in COMP 310
- Abstraction for how data can be exchanged across the network.
  - Connections between processes on (different) computers.
- 3 classes of Internet socket: UDP, TCP, Raw.
- Sockets:
  - Addresses (IP addresses)
  - Port numbers (integers)



# Client-server connection (jabber)



## Assignment: 206bot

- Robot that sits on your computer.
- Reads commands periodically.
- Allows commands to be sent to it.
- Basis of a botnet, but also a file sharing service, a system for distributed computing, etc.



- 1) Open a socket to an "evil master controller" to register its availability. The particular master controller is to be randomly selected from a list of alternative ones to provide robustness.



- The choices are:
  http://quintessence.cim.mcgill.ca:8080/206/register
  http://epitome.cim.mcgill.ca:8080/206/register
  http://www.aquarobot.net:8080/206/register
- Registration implies loading the URL ones of the addresses above, along with completed fields for isfrom (your name), host and port. For example:
- http://epitome.cim.mcgill.ca:8080/206/register?isfrom=Greg&host=localhost&port=9999



- 2) open a network connection (a TCP network socket), number 7551, on your machine.
- Use it to accept and execute command messages from the master controller that tell your machine what to do.
- In general, this kind of communication method would allow for instantaneous control of the bots, but might not work if your machine is behind a firewall that restricts access to port 7551.

- At regular intervals, for the assignment this is every 2 minutes, fetch and run a command file from the "evil master controller."



## Protocol

- getpage
- execute
- echo
- xyzzy
- none
- bulletins
- surprise
- store
- master



- Sample code

- http://www.cim.mcgill.ca/~dudek/bot.zip

- Specification:
- http://www.cim.mcgill.ca/~dudek/botspec.txt
-



## Cryptographic signing

- Public key security
- Public key & private key
- Hashing

## Like the bot assignment?

- When the course is done, talk to be about this if you want to be part of a team developing this app for a couple of weeks.



## What is CGI (review)

- CGI: Common Gateway Interface
- A set of simple rules for connecting an application to a web server
  - What's a web server? The program that provides web pages from your computer's file system to clients on the internet.
    - » Apache (open source!)
    - » Microsoft IIS
      - » Mac OS (now uses Apache)
    - » Zope (www.zope.org, more than just a server)
    - » etc…



## CGI script structure

- Check form fields
  - use cgi.FieldStorage class to parse query
    - » takes care of decoding, handles GET and POST
    - » "foo=ab+cd%21ef&bar=spam" -->
      {'foo': 'ab cd!ef', 'bar': 'spam'} # (well, actually, ...)
- Perform action
  - this is up to you!
  - database interfaces available
- Generate HTTP + HTML output
  - (HTTP is the way to returning data, HTML is the formatting.)
  - print statements are simplest
  - template solutions available



## Structure refinement

```
form = cgi.FieldStorage()
if not form:
    ...display blank form...
elif ...valid form...:
    ...perform action, display results (or next form)...
else:
    ...display error message (maybe repeating form)...
```

## FieldStorage details

- Behaves *like* a dictionary:
  - .keys(), .has_key()  # but not all other methods!
  - dictionary-like object ("mapping")
- Items
  - values are MiniFieldStorage instances
    - » **.value** gives field value!
  - if multiple values: *list* of MiniFieldStorage instances
    - » if type(...) == types.ListType: ...
  - may also be FieldStorage instances
    - » used for file upload (test .file attribute)



## Other CGI niceties

- cgi.escape(s)
  - translate "<", "&", ">" to "<", "&", "&gt"
- cgi.parse_qs(string, keep_blank_values=0)
  - parse query string to dictionary {"foo": ["bar"], ...}
- cgi.parse([file], ...)
  - ditto, takes query string from default locations
- urllib.quote(s), urllib.unquote(s)
  - convert between "~" and "%7e" (etc.)
- urllib.urlencode(dict)
  - convert dictionary {"foo": "bar", ...} to query string "foo=bar&..." # note asymmetry with parse_qs() above



## Dealing with bugs

- Things go wrong, you get a traceback...
- By default, tracebacks usually go to the server's error_log file…
- Printing a traceback to stdout is tricky
  - could happen before "Content-type" is printed
  - could happen in the middle of HTML markup
  - could contain markup itself
- What's needed is a...



## Debugging framework

```
import cgi

def main():
    print "Content-type: text/html\n" # Do this first
    try:
        import worker          # module that does the real work
    except:
        print "<!-- --><hr><h1>Oops.  An error occurred.</h1>"
        cgi.print_exception() # Prints traceback, safely

main()
```

## Security notes

- CGI scripts need guard against malicious entry.
- Watch out when passing fields to the shell
  - e.g. os.popen("finger %s" % form["user"].value)
  - what if the value is "; cat /etc/passwd" ...
- Example (imperfect) solutions:
  - Quote:
    - » user = pipes.quote(form["user"].value)
  - Refuse:
    - » if not re.match(r"^\w+$", user): ...error...
  - Sanitize:
    - » user = re.sub(r"\W", "", form["user"].value)



## Multi-step interactions

- HTTP is "stateless"
  - Each page/web request is independent.
  - There is no natural notion of the next interaction or the last one.
    - » When a request arrives, it could be from the same person who made the previous one, or maybe not.
- An *connected set of interactions* must somehow implement
  - persistence (information that is remembered)
  - identity (an ID to something like it)
- Approaches: manually, cookies, hidden form fields, URL encoding.



## More state: Trivial (bad) idea

- Continuity via ID
- On each web form, include 2 manual fields
  - ID number
  - step number (in a series of steps)
    - » e.g. first register, then pick and item, then fill in credit card, then fill in shipping addres, then …
- Problem: don't want to have to fill this out repeatedly,
- Problem: could lie (too easily).



## Automatic transfer of ID

- Use fact that form fields are automatically sent to sever when forms are submitted.
- The server can pre-fill fields that hold state information (like your ID).
- These can, further, be hidden from the user
  - to make it more attractive
  - to reduce chances of tampering

# Session maintenance

Correlate requests from same user

- Assign session key on first contact
- Incorporate session key in form or in URL

Options:

1. In form: use hidden input field:
   1. <input type="hidden" name="session" value="1f9a2">
2. In URL:
   - » http://myhost.com/cgi-bin/myprog.py/1f9a2
   - » passed in environment (os.environ[...]):
     - » PATH_INFO=/1f9a2
     - » PATH_TRANSLATED=<rootdir>/1f9a2



# Extra python ideas & review & reminders

- module **os** provides operating system functions (e.g. most system calls).
  - read, write, seek, etc.,… In a ***portable*** *format!*
- module **sys** provides interpreter/context info.
  - stdin, stdout, stderr
- module **string** provides string manipulation.
  - Python version 2 and later makes these methods part of all string automatically, but read the documentation for module string.
    - » `string.split("hello,there",",")` or `"hello,world".split(",")`
- find docs `using __doc__`
  - find list of things in a module using `__dict__ (very cryptic)`