The *formerly* Informal Python Boot Camp

http://emolch.org/pythonbootcamp.html

by Sebastian Heimann

Lesson 1: Introduction to basic Python

Zoology of programming languages

Reptile show

Primitives

Containers

Flow control

Functions

Input and output

Things to remember from today's lesson

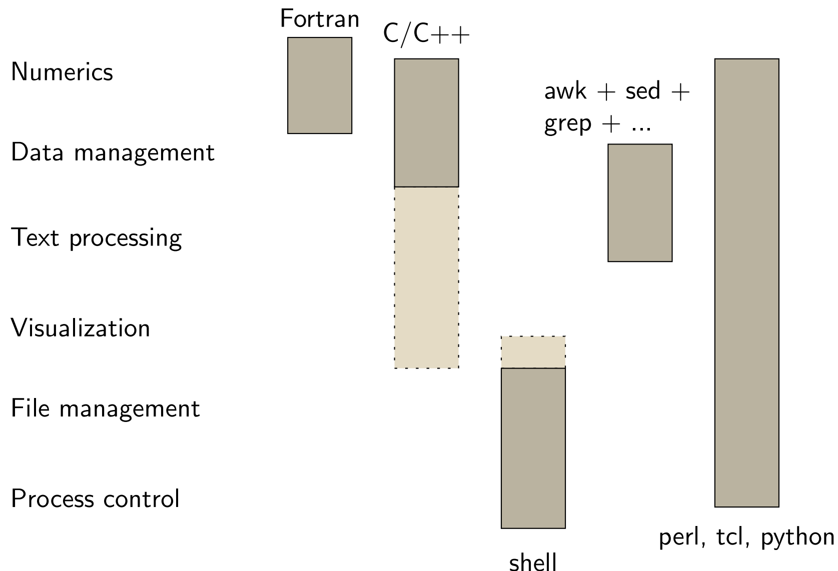
Further reading

Different tasks

As scientists, we get excited about a whole range of different computational tasks:

- Numerics
- Data management
- Text processing
- Visualization
- File management
- Process control

Different tasks - different tools



The Unix philosophy

Write tools that do one thing and do it well.
Write tools to work together.

- Small programs tied together with shell scripts.
- Library functions tied together by a short main program.
- Library functions and external programs tied together by a high level script.

Some popular high level script/programming languages

- **Lisp** (traditional)
- **Perl** (1987)
- **Tcl** (1988)
- **Python** (1991)
- **Dylan** (early 1990s)
- **Java** (1995)
- **Ruby** (1995)

Why are these languages so popular? (1)

Less code = less errors

And faster development, quicker understanding, faster typing, faster finding errors, better overview, easier to modify.

Why are these languages so popular? (2)

More like brain language

Easier to develop, understand, maintain, remember, ...

Why are these languages so popular? (3)

No (separate) compilation step

- No compiler problems
- No makefiles
- No linker problems
- Faster development cycles

Why are these languages so popular? (4)

Great standard libraries included

- Platform independence
- One place to look first for a proven solution
- Reuse instead of reinvent

Inventor of Python

Guido van Rossum

http://www.python.org/~guido/

Python features

- ”Very high level language”
- Clean, spare syntax
- Strong, dynamic typing
- Object-oriented, multi-paradigm
- Garbage collection

The Zen of Python, by Tim Peters

```
% python
>>> import this
```

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren’t special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
...

Example: See if there is Bratwurst in the mensa today

STUDIERENDENWERK
HAMBURG

Speisekarte

	Mittwoch, 05.09.2007	Preise**	
		Studierende	übrige Gäste
Beliebt und gerne gegessen	**Indisches Tandoori Chicken (1,20) mit Krautsalat (3,5) und Fladenbrot als Beilage (14)**	**1,80 €**	**2,75 €**
Die Alternative	**Ägyptische Falafel auf Minz-Joghurt-Soße (20) dazu Reis und Krautsalat (3,5)**	**2,00 €**	**3,00 €**
Campus Spezial	**Lammkeule in Thymian-Rotweinsoße (1,4,11,14,16,20,22,23) dazu Princessböhnchen natur und Kartoffeltaler (1,14,16,19,20,22)**	**3,35 €**	**3,95 €**

= Fleischloses Gericht

= mit Schweinefleisch

= mit Alkohol

*) Bio-Gericht aus biologischem Anbau bzw. Aufzucht

Example: See if there is Bratwurst in the mensa today

```
import urllib, os

# the daily mensa menu page
url = 'http://www.studentenwerk-hamburg.de' + \
      '/essen/tag.php?haus=Geomatikum'

# get the web page
page = urllib.urlopen( url ).read()

# check for availability of bratwurst
if page.lower().find('bratwurst') != -1:
    # tell everybody
    wall = os.popen( 'wall', 'w' )
    wall.write("wurst! wuuuurst!!!")
```

This course will now loosely follow the official Python Tutorial:
An Informal Introduction to Python:

- `http://docs.python.org/tut/`

Primitives

Interactive Python - Numbers - Strings

Interactive Python

You can use python interactively.

```
> python
Python 2.5.1 ...
>>> 1+1
2
>>> print "1 + 1 =", _
1 + 1 = 2
```

Python as a calculator

```
>>> from math import *  # to get math functions
>>> sin(pi/2.0)
1.0
```

Integer division returns floor():

```
>>> 5/2
2
>>> 5/-2
-3          # unlike in C!
```

Arithmetic type casting as usual:

```
>>> 5.0/2
2.5
>>> 5/2.0
2.5
```

Complex numbers

Complex arithmetic is built right into the language:

```
>>> 1j**2
(-1+0j)
>>> a = (3+4j)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a)
5.0
```

Variables

A local variable is created simply by assigning it a value:

```
>>> earth_radius = 6.371e6
>>> print earth_radius * 2.
12742000.0
```

Note: Technically, instead of assigning a number to a variable, the identifier `earth_radius` is bound to an immutable number object here. This is a subtle difference to other languages and we must revisit this issue later!

Strings

Strings are defined with single or double quotes:

```
>>> print 'Indisches Tandoori Chicken'
Indisches Tandoori Chicken
>>> print "Indisches Tandoori Chicken"
Indisches Tandoori Chicken
```

Strings

Escape characters work as in other languages:

```
>>> print '\'\''
''
>>> print 'X\nX'
X
X
>>> print '\\'
\
```

Long strings

Triple quotes may be used to define a string spanning several lines:

```
>>> usage = """Usage: attack [OPTIONS] host ...
...     -a  Try all attacks
...     -v  Verbose
... """
```

Basic string operations

Concatenation:

```
>>> 'sp' + 'am'
'spam'
>>> 'spam' * 10
'spamspamspamspamspamspamspamspamspamspam'
```

Basic string operations

Substrings:

```
>>> food = 'bratwurst'
>>> food[0]
'b'
>>> food[0:1]
'b'              # different than in other languages!
>>> food[1:4]
'rat'
>>> food[-5:]
'wurst'
```

Think like this, when using slices:

0	1	2	3	4	5	6	7	8	9
b	r	a	t	w	u	r	s	t	
-9	-8	-7	-6	-5	-4	-3	-2	-1	

Strings (and numbers) are immutable

Strings in Python cannot be changed!

```
>>> food = 'bratwurst'
>>> food[0:4] = 'curry'        # impossible!
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support slice assignment
```

But, creating a new string is easy and efficient:

```
>>> food = 'bratwurst'
>>> food = 'curry' + food[-5:]
>>> print food
currywurst
```

String methods

Strings are objects with many useful methods:

```
>>> food = 'bratwurst'
>>> food.find( 'wurst' )  # search for substring
4
```

```
>>> food = '  bratwurst  '
>>> food.lstrip()  # remove leading whitespace
'bratwurst  '
>>> food.strip()   # remove whitespace at both ends
'bratwurst'
```

More string methods

```
>>> 'Indisches Tandoori Chicken'.split()
['Indisches', 'Tandoori', 'Chicken']
```

```
>>> ' und '.join(["Schnitzel", "Pommes", "Salat"])
'Schnitzel und Pommes und Salat'
```

There are more useful string methods like `startswith`, `endswith`, `lower`, `upper`, `ljust`, `rjust`, `center`, ...
Always have an eye on the Python Library Reference when doing something with strings:

- `http://docs.python.org/lib/string-methods.html`

String length

Use `len(string)` to get the length of a string:

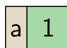
```
>>> len('bratwurst')
9
```

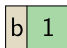
Similar but different

Variables vs. identifiers

In C and Fortran, a variable behaves like a box containing a value:

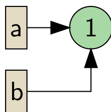
```
a = 1;
b = a;
```





Python, in contrast, has names (or *identifiers*), which can be bound to objects:

```
a = 1
b = a
```



Variables vs. identifiers

Example: add two numbers

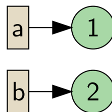
```
a = 1
b = 2
b = a + b
```

Let’s have a very close look at this primitive operation...

Variables vs. identifiers

(step 1)

```
a = 1
b = 2
```

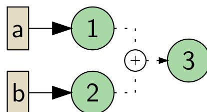


We start with two identifiers which are bound to two different number objects...

Variables vs. identifiers

(step 2)

```
a = 1
b = 2
    a + b
```

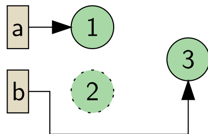


The `+` operator creates a new number object...

Variables vs. identifiers

(step 3)

```
a = 1
b = 2
b = a + b
```



Finally, the identifier `b` is rebound to the new number object.

Containers

Lists - Tuples - Dictionaries

Lists

A list is created with brackets `[]`:

```
>>> foods = [ "Schnitzel", "Pommes", "Salat" ]
```

List elements may be of different types:

```
>>> print [ 'One', 2, 3.0, foods ]
['One', 2, 3.0, ['Schnitzel', 'Pommes', 'Salat']]
```

Empty list:

```
>>> empty = []
```

Lists

Lists, like strings, are *sequence types*.
So, many things work the same:

```
>>> foods = ['Schnitzel', 'Pommes', 'Salat']
>>> len(foods)
3
>>> foods[0:2]
['Schnitzel', 'Pommes']
```

Modifying lists

But unlike strings (which are *immutable*), lists are of *mutable sequence type*.

```
# Replace first two elements
>>> foods[0:2] = [ "Bratwurst", "Mayo" ]
>>> print foods
['Bratwurst', 'Mayo', 'Salat']

# Remove last two elements
>>> foods[-2:] = []
>>> print foods
['Bratwurst']
```

Modifying lists

```
# Insert new elements at beginning
>>> foods[0:0] = [ "Bier", "Korn" ]
>>> print foods
['Bier', 'Korn', 'Bratwurst']

# But ...
>>> foods[0] = [ "Bier", "Korn" ]
>>> print foods
[['Bier', 'Korn'], 'Korn', 'Bratwurst']
```

Modifying lists

Assignment does not copy a list:
(Identifiers are references to objects...)

```
>>> a = [1,2,3,4]
>>> b = a                        # a and b now refer to
                                 # the same list object!
>>> b[3] = 'surprise!'
>>> print a
[1, 2, 3, 'surprise!']
>>> print b
[1, 2, 3, 'surprise!']
```

Modifying lists

Slicing always copies:

```
>>> a = [1,2,3,4]
>>> b = a[:]                # b is now a (shallow)
                            # copy of a!
>>> b[3] = 'surprise!'
>>> print a
[1, 2, 3, 4]
>>> print b
[1, 2, 3, 'surprise!']
```

List methods to remember

- `append(x)`: Add `x` to the end.
- `extend(L)`: Append all elements of `L`.
- `insert(i,x)`: Insert `x` at position `i`.
- `pop()`: Remove and return last element.
- `pop(i)`: Remove and return element at position `i`.
- `remove(x)`: Remove first element whose value is `x`.
- `index()`: Return index of first element whose value is `x`.
- `sort()`: Sort list.
- `count(x)`: Return the number of times `x` appears.
- `reverse()`: Reverse order of elements.

Using lists as stacks

A list used as a *last in, first out* data structure:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack
[3, 4, 5]
```

Using lists as queues

A list used as a *first in, first out* data structure:

```
>>> queue = ["Eric", "John", "Michael"]
>>> queue.append("Terry")       # Terry arrives
>>> queue.append("Graham")      # Graham arrives
>>> queue.pop(0)
'Eric'
>>> queue.pop(0)
'John'
>>> queue
['Michael', 'Terry', 'Graham']
```

List comprehensions

List comprehensions are used to create lists, *in place* based on the values of other lists:

```
>>> vec = [2, 4, 6]

>>> [3*x for x in vec]
[6, 12, 18]

>>> [3*x for x in vec if x > 3]
[12, 18]

>>> [3*x for x in vec if x < 2]
[]

>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
```

Tuples

Tuples are immutable lists.

- Round parantheses create tuples.
- Parantheses can be ommited in many cases.

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

Tuples

Creating tuples with zero or one element:

```
>>> empty = ()
>>> singleton = ('hello',)    # <-- trailing comma!
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

Tuples

Tuples are often used to pass around grouped values.

Packing and unpacking of tuples:

```
>>> t = (1,2,3)      # pack
>>> (a,b,c) = t      # unpack
>>> print a,b,c
1 2 3

>>> a,b = a*2, b*2  # multiple assignment!
>>> b,a = a,b        # swap values!
```

It's also OK to unpack a list into a tuple:

```
>>> (a,b,c) = [4,5,6]
>>> print a,b,c
4 5 6
```

Dictionaries

Dictionaries are used to store *key/value* pairs.

- Curly braces create dictionaries.
- Contents of a dictionary are unordered.
- Keys are unique.

```
>>> colors = {'red': (1, 0, 0),
...           'green': (0, 1, 0)}
>>> colors['red']                    # lookup value
(1, 0, 0)
>>> colors['blue'] = (1, 0, 0)  # add new element
>>> colors
{'blue': (0, 0, 1), 'green': (0, 1, 0),
 'red': (1, 0, 0)}
```

Dictionaries

Two ways to check if the dictionary has a specific key:

```
>>> colors.has_key( 'red' )
True
>>> 'red' in colors
True
```

Delete an element from the dictionary:

```
>>> del colors['red']
>>> colors
{'blue': (0, 0, 1), 'green': (0, 1, 0)}
```

Dictionaries

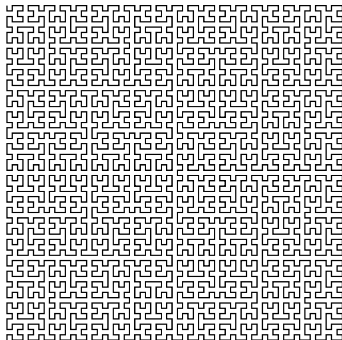
To construct a dictionary from a list of tuples:

```
>>> dict([('seismology', 254),
...       ('seismics', 250),
...       ('vulcanology', 253)])
{'vulcanology': 253, 'seismics': 250,
 'seismology': 254}
```

A shortcut, when the keys are simple strings:

```
>>> config = dict( verbose=True,
                   recursive=False,
                   inputfile='data.txt' )
>>> config
{'inputfile': 'data.txt', 'verbose': True,
 'recursive': False}
```

Flow control



Executable Python scripts

print_args.py:

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

print sys.argv
```

```
> python print_args.py 1 2 hallo
['print_args.py', '1', '2', 'hallo']
> chmod +x print_args.py
> ./print_args.py 2 3 bye
['./print_args.py', '2', '3', 'bye']
```

Indentation

Python uses indentation to mark code blocks!

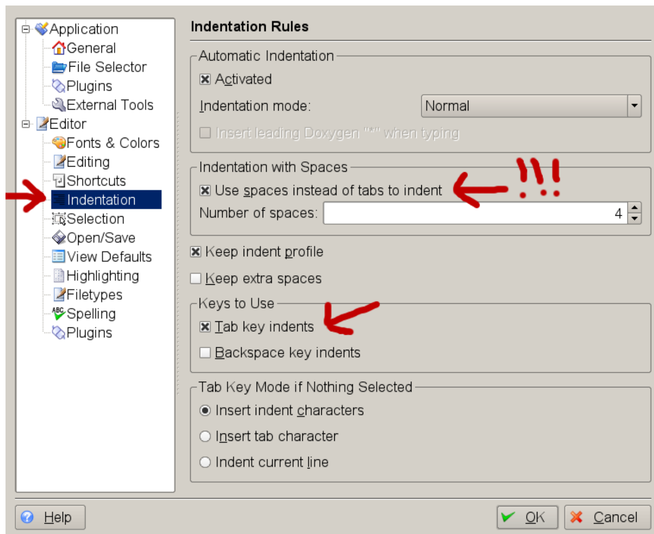
```
x = int(raw_input('gimme an int'))
if x < 23:
    print 'x is less than 23'
    if x < 5:
        print 'it is even less than 5!'
        print 'but who cares?'

print 'the value of x is:', x
```

Indentation

- Don’t mix tabs an spaces.
- Python looks at the exact whitespace characters in your source.
- Your editor doesn’t show you if there’s a tab or 8 spaces, but it matters to python.
- Your editor can be set to use only spaces, even when you press TAB.

Leash your editor...



If statements

```
x = int(raw_input("Please enter an integer: "))
if x < 0:
     x = 0
     print 'Negative changed to zero'
elif x == 0:
     print 'Zero'
elif x == 1:
     print 'Single'
else:
     print 'More'
```

What is true - what is false

- Non-zero integers are *true*.
- Zero is *false*.
- Any sequence with a non-zero length is *true*.
- Empty sequences are *false*.
- `True` is *true*.
- `False` is *false*.
- `None` is *false*.

```
import sys
files  = sys.argv[1:]
if not files:
    # missing command line arguments
    print "usage: dosomething.py filename ..."
    sys.exit(1)

# do something with files...
```

While statement

Let’s see how far we can count in Python:

```
>>> import time
>>> i = 1
>>> while True:
...     i = i * 1000   # same as: i *= 1000
...     print repr(i)
...     time.sleep(1)  # wait one second
...
1000
1000000
1000000000
1000000000000L   # <- type conversion occured!
1000000000000000L
# ... continues until memory is exhausted!
```

For statements

The `for` statement iterates over the items of any sequence:

```
menu = ['Schnitzel', 'Pommes', 'Eis', 'Salat']
for food in menu:
    print food

for letter in 'abc':
    print letter
```

```
Schnitzel
Pommes
Eis
Salat
a
b
c
```

For statements

But don’t change the list you are iterating over!

Use a copy of the list in case:

```
>>> menu = ['Schnitzel', 'Pommes']
>>> for food in menu[:]:
...     menu.append( food )
>>> menu
['Schnitzel', 'Pommes', 'Schnitzel', 'Pommes']
```

The range function

Range creates a list of numbers:

```
>>> for i in range(4):
...     print i, 2**i
...
0 1
1 2
2 4
3 8
```

The range function behaves similar to slicing operators:

```
>>> range(4)
[0, 1, 2, 3]
>>> range(3,7)
[3, 4, 5, 6]
>>> range(2,10,2)
[2, 4, 6, 8]
```

The continue statement

The `continue` statement brings you to the next iteration:

```
import sys
files  = sys.argv[1:]
for file in files:
    if file[-4:] != '.jpg':    # skip non-jpeg files
        continue

    # do something with the jpeg file
    blablabla(file)
```

The break statement

The `break` statement let's you abort a loop abnormally:

```
good_foods = [ 'Schnitzel', 'Pommes' ]
menu = ['Schnitzel', 'Pommes', 'Eis', 'Salat']
wanted = []
for food in menu:
    if food in good_foods:
        wanted.append( food )

    if len(wanted) == 2:
        break  # two things are enough today...

print "ich will: " + ' '.join(wanted)
```

The pass statement

The `pass` statement does nothing:

```
>>> while True:
...     pass # Busy-wait for keyboard interrupt
...
```

It can be used when a statement is required syntactically but the program requires no action.

Looping techniques

To enumerate items of a sequence:

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print i, v
...
0 tic
1 tac
2 toe
```

Looping techniques

To iterate over several sequences simultaneously:

```
>>> for a,b,c in zip(['ene', 'raus'],
                     ['mene', 'bist'],
                     ['mu', 'du']):
...     print a, b, c
...
ene mene mu
raus bist du
```

Looping a dictionary

Efficient:

```
>>> knights = {'gallahad': 'the pure',
               'robin': 'the brave'}
>>> for k, v in knights.iteritems():
...     print k, v
...
robin the brave
gallahad the pure
```

Sorted by keys:

```
keys = knights.keys()
keys.sort()
for k in keys:
    print k, knights[k]
```

Looping a dictionary

Sorted by values:

```
def by_value(a,b):
    return cmp(knights[a],knights[b])

keys = knights.keys()
keys.sort(by_value)
for k in keys:
    print k, knights[k]
```

Functions

Defining functions

The `def` keyword starts a function definition:

```
>>> def powers_of_two(n):
...     results = []
...     for i in range(n):
...         results.append( 2**i )
...     return results
...
>>> print powers_of_two(10)
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

Default argument values and keyword arguments

Giving a default value makes a parameter optional

```
def ask_ok( prompt, retries=4,
            complaint='Yes or no, please!'):
    while True:
        ok = raw_input(prompt)
        if ok in ('y', 'ye', 'yes'): return True
        if ok in ('n', 'no'): return False
        retries = retries - 1
        if retries < 0:
            raise IOError, 'refusing user'
        print complaint

ask_ok('Do you really want to quit?')
ask_ok('OK to overwrite the file?', 2)
ask_ok('Coffee?', complaint='Yes or no, dude!')
```

Arbitrary argument lists

You can allow and catch any additional arguments:

```
import os

def cautious_delete( prompt, *filenames ):
    for file in filenames:
        if 'y' == raw_input(prompt % file):
            os.remove(file)

cautious_delete( "Delete file '%s'? [y/N] ",
                      "lesson1.pdf", "lesson2.pdf" )
```

```
Delete file 'lesson1.pdf'? [y/N] y
Delete file 'lesson2.pdf'? [y/N] y
```

Arbitrary keyword arguments

Functions may also take additional keyword arguments:

```
def eat( **howmany_of ):
    for food in howmany_of.keys():
        print ("I will eat %i %s today!"
                 % (howmany_of[food], food))

eat( apples=10, bananas=13, spagettis=333 )
```

```
I will eat 333 spagettis today!
I will eat 10 apples today!
I will eat 13 bananas today!
```

The keyword arguments are put in a *dictionary*.
More about *dictionaries* later.

Unpacking argument lists

Sometimes you have a function taking several arguments:

```
delete_files( 'lesson1.pdf', 'lesson2.pdf' )
```

but you have the intended arguments in a list:

```
files = [ 'lesson1.pdf', 'lesson2.pdf' ]
# doesn't work:
delete_files( files )
# doesn't always work:
delete_files( files[0], files[1] )
```

In these cases you can use *argument unpacking*:

```
delete_files( *files )
```

Documentation strings

```
def my_function():
    """Do nothing, but document it.

    No, really, it doesn't do anything.
    """
    pass

print my_function.__doc__

help(my_function)
```

1. Brief summary (Start with capital letter, end with period.)
2. Blank line
3. Complete description

Input and output variables

Functions can return more than one value:

```
def spherical_coords( x, y, z ):
    # ...
    return r, theta, phi

r, theta, phi = sperical_coords( x, y, z )
```

Call by value (C)

In C we have *call by value*:

```
void f(int j) {
    j++;
    printf("inside: %i\n", j);
}

main() {
    int i;
    i = 1;
    printf("before: %i\n", i);
    f(i);
    printf("after:  %i\n", i);
}
```

```
before: 1
inside: 2
after:  1
```

Call by reference (C, with pointer)

In C we can use pointers to do a *call by reference*:

```
void f(int *j) {
    (*j)++;
    printf("inside: %i\n", *j);
}

main() {
    int i;
    i = 1;
    printf("before: %i\n", i);
    f(&i);
    printf("after:  %i\n", i);
}
```

```
before: 1
inside: 2
after:  2
```

Call by reference (C++)

C++ has built in an extra *call by reference*:

```
void f(int &j) {
  j++;
  cout << "inside: " << j << endl;
}

main() {
  int i;
  i = 1;
  cout << "before: " << i << endl;
  f(i);
  cout << "after:  " << i << endl;
}
```

```
before: 1
inside: 2
after:  2
```

Call by reference (Fortran)

Fortran always does *call by reference*:

```
subroutine f(j)
    integer :: j
    j = j+1
    print *, "inside:", j
end subroutine

program reference
    integer :: i = 1
    print *, "before:", i
    call f(i)
    print *, "after: ", i
end program
```

```
before: 1
inside: 2
after:  2
```

Call by object reference

Python, seems to do *call by value*...

```
def f(j):
    j += 1
    print "inside:", j

i = 1
print "before:", i
f(i)
print "after: ", i
```

```
before: 1
inside: 2
after:  1
```

... but this is not the case!

Inside the function, j is rebound to a new number object with the value 2. Identifier i is not rebound.

Call by object reference

Python is passing *object references by value*.
Not the value of the object.

```
def f(j):
    j[0] += 1
    print "inside:", j[0]

i = [ 1 ]
print "before:", i[0]
f(i)
print "after: ", i[0]
```

```
before: 1
inside: 2
after:  2
```

Input and output

Opening a file

Use `open(filename,mode)` to open a file:

```
f = open('/tmp/workfile', 'w')
```

Some possible modes:

- `r`: Open text file for read.
- `w`: Open text file for write.
- `a`: Open text file for append.
- `rb`: Open binary file for read.
- `wb`: Open binary file for write.

Open returns a *File Object*.

To close the file, use:

```
f.close()
```

Predefined File Objects

- `sys.stdin`: Standard input
- `sys.stdout`: Standard output
- `sys.stderr`: Standard error

Consequent use of `stderr` for all warnings, status messages and error messages while using `stdout` for any ”real” output is a good practice, because this allows cool piping tricks!

Reading from a File Object

Read a quantity of data from a file:

```
s = f.read( size )  # size: number of bytes to read
```

Read entire file:

```
s = f.read()
```

Read one line from file:

```
s = f.readline()
```

Get all lines of data from the file into a list:

```
list = f.readlines()
```

Iterate over each line in the file:

```
for line in f:
    print line,
```

Writing to a File Object

Write a string to the file:

```
f.write( string )
```

Write several strings to the file:

```
f.writelines( sequence )
```

Example: A primitive `cat` replacement

```python
import sys

if len(sys.argv) > 1:
    for filename in sys.argv[1:]:
        f = open(filename, 'r')
        for line in f:
            sys.stdout.write(line)

        f.close()
else:
    sys.stdout.writelines( sys.stdin )
```

```
Usage: cat.py [FILE]...
Concatenate FILE(s), or standard input,
to standard output.
```

Inter-process communication with pipes

Read from another program:

```
import os
gz = os.popen('gunzip -c compressed_file.gz','r')
uncompressed_data = gz.readlines()
gz.close()
```

Write to another program:

```
import os
gz = os.popen('gzip > compressed_file.gz','w')
gz.write(uncompressed_data)
gz.close()
```

The `pickle` module

Saving of arbitrary python data structures is trivial:

```
>>> import pickle
>>> x = ['a', ['nested', 'data', 'structure',
         (1, 2, 3)]]

>>> f = open("my_file", "w")
>>> pickle.dump( x, f )
>>> f.close()

>>> f = open("my_file", "r")
>>> x =  pickle.load( f )
>>> f.close()

>>> print x
['a', ['nested', 'data', 'structure', (1, 2, 3)]]
```

See also these other modules: `shelve`, `anydbm`, `cPickle`

Things to remember from today's lesson

- Others have variables.
- Python has identifiers (which are references to objects).
- Strings and numbers are *immutable*.
- Lists are *mutable*.
- Strings and lists are *sequence types* and behave similar.
- The stairways to power are lists and dictionaries.

Keep these pages open, when programming in Python:

- Tutorial:
  `http://docs.python.org/tut/`
- Library Reference:
  `http://docs.python.org/lib/lib.html`

Further reading

- Good introduction to ”Pythonic” programming
  `http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html`
- Coding conventions for Python
  `http://www.python.org/dev/peps/pep-0008/`
- Alex Martelli giving a Google tech talk ”Python for programmers”
  `http://www.youtube.com/watch?v=ujkzfC2lebA`