

# Introduction to Python

Demitri Muna  
NYU

22 June 2010

# Introduction to Python

- No experience with Python is necessary, but we're assuming you've written programs before.
- Using Python 2.6 or higher. Can test your Python version with:

```
% python --version
```

- Python 3.0 is out. It breaks some old code (not much), but most people are still on 2.6.
- Language is continually being updated and modified. More libraries are being added, both in the language and by third parties.
- Try out the examples as we go through them.

# Hello World

The simplest application:

I left space to explain the code, but...

Run as:

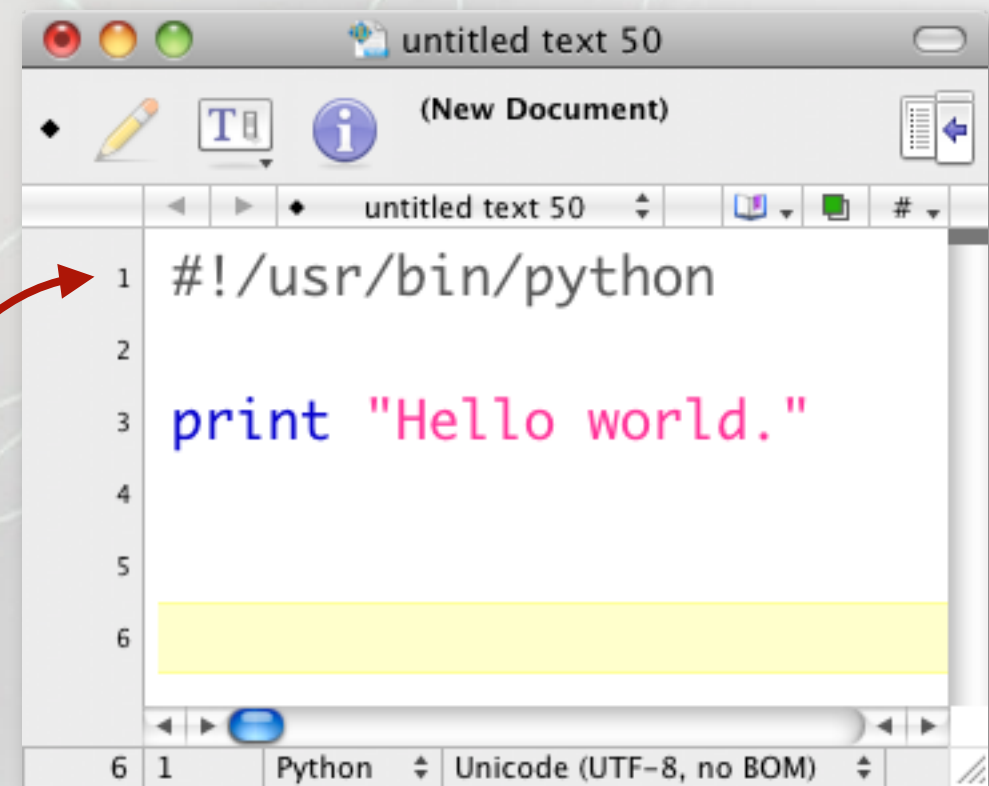
```
% python hello_world.py
```

or, make it an executable:

```
% chmod +x hello_world.py
```

```
% hello_world.py
```

tells OS what to run  
the program with



# Numbers

Assigning variables, familiar syntax.

## Numeric types

integer

long integer

octal (base 8)

decimal

complex

“long” integers  
can be any length!

Don't write numbers with leading zeros --  
they become octal!

Append a “j” to a number to make it  
complex (engineers use “j”, physicists use “i”  
for  $\sqrt{-1}$  ).

Note: this  
behavior will  
change in the  
future (see  
truncating  
division).

```
1 #!/usr/bin/python
2
3 # numbers
4 a = 42
5 b = 12 + 45
6
7 # numeric types
8 c = 3
9 d = 3L
10 e = 027
11 f = 027.
12 g = 10j
13 h = complex(3,5)
14 print h.real, h.imag
15
16 print 10/3
```

The screenshot shows a text editor window with the following code and annotations:

- Line 3: `# numbers` - A red arrow points from the text “comment in Python” to this line.
- Line 4: `a = 42` - A red arrow points from the text “integer” to this line.
- Line 5: `b = 12 + 45` - A red arrow points from the text “long integer” to this line.
- Line 8: `c = 3` - A red arrow points from the text “octal (base 8)” to this line.
- Line 9: `d = 3L` - A red arrow points from the text “decimal” to this line.
- Line 12: `g = 10j` - A red arrow points from the text “complex” to this line.
- Line 16: `print 10/3` - A red arrow points from the text “Note: this behavior will change in the future (see truncating division).” to this line.



# Numbers

Python operators:

+	-	*	**	/	//	%
<<	>>	&		^	~	bitwise operators
<	>	<=	>=	==	!=	<>

Annotations:  
- **exponent** points to \*\*  
- **truncating division** points to //  
- **modulo** points to %  
- **same, but only use !=** points to !=  
- **bitwise operators** points to the row containing <<, >>, &, |, ^, ~

`import`

This command makes an external package available for additional functionality. This one is built into python.

Note the format of `moduleName.value` (or function)

(This keeps the runtime light since you are only loading the functionality that you use.)

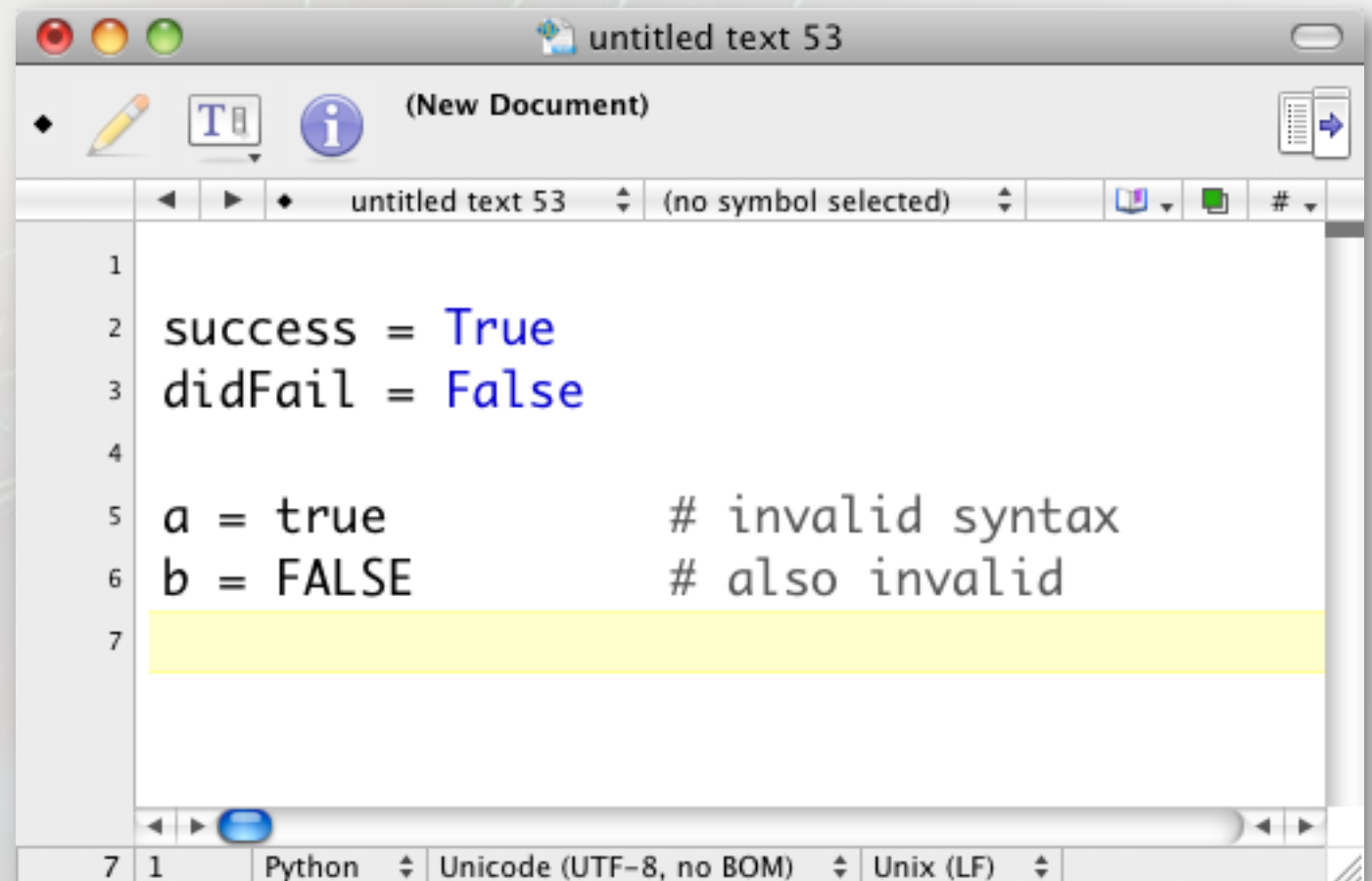
```
1 #!/usr/bin/python
2
3 import sys
4
5 # largest integer number on this machine
6 print sys.maxint
7
8 # smallest integer on this machine
9 print -sys.maxint - 1
10
```

You will get a different result running on a 32-bit vs a 64-bit machine (something to be aware of when running your code in different places.)

# Boolean Values

Boolean values (True/False) are native types in Python.

The capitalization is important.



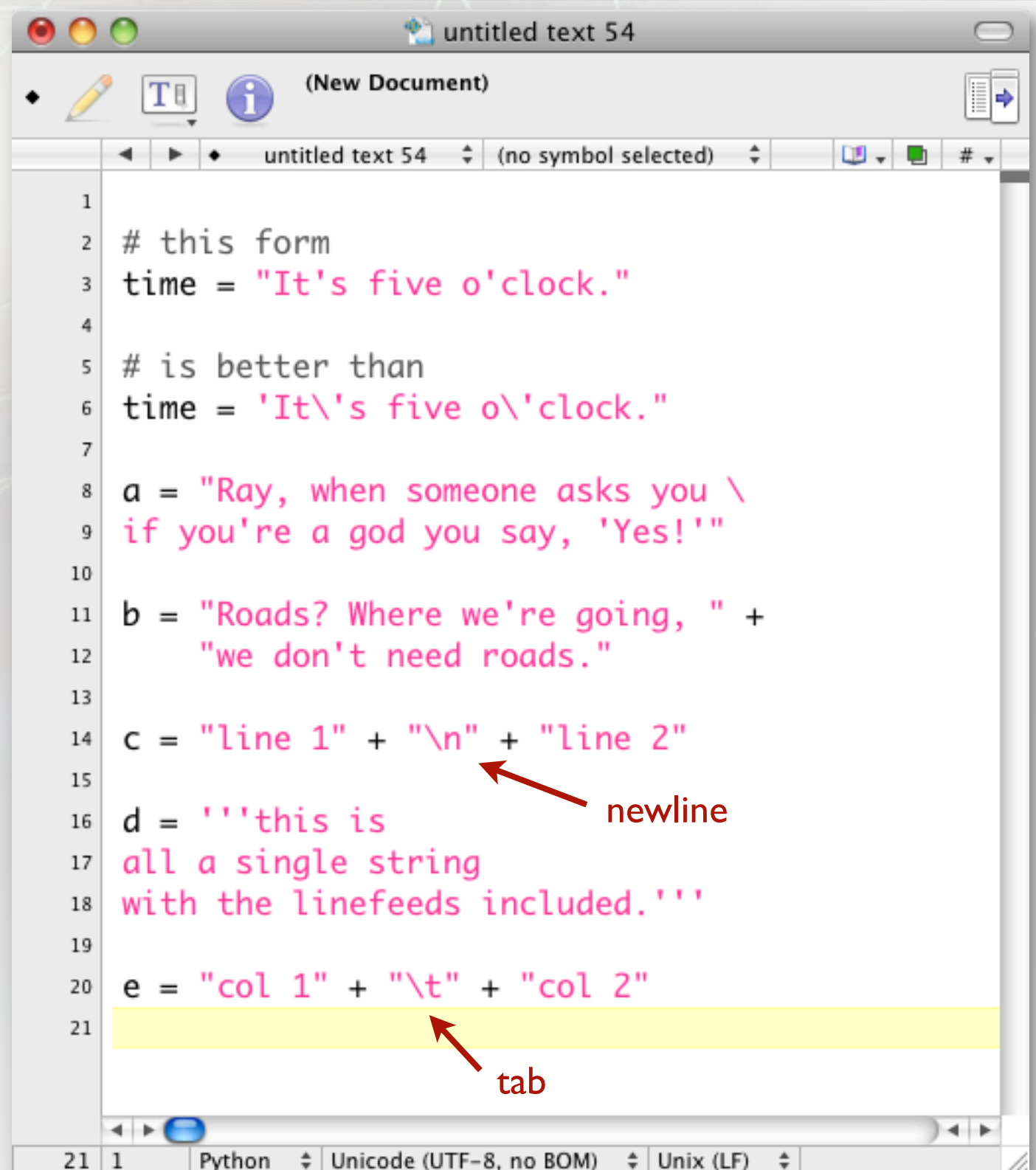
```
1  
2 success = True  
3 didFail = False  
4  
5 a = true           # invalid syntax  
6 b = FALSE          # also invalid  
7
```

# Strings

Strings can be delimited using single quotes, double quotes, or triple quotes. Use whatever is convenient to avoid having to escape quote characters with a “\”.

Strings can be joined together with the “+” operator.

Triple quotes are special in that they let you span multiple lines. Can be three single quotes or three double quotes.



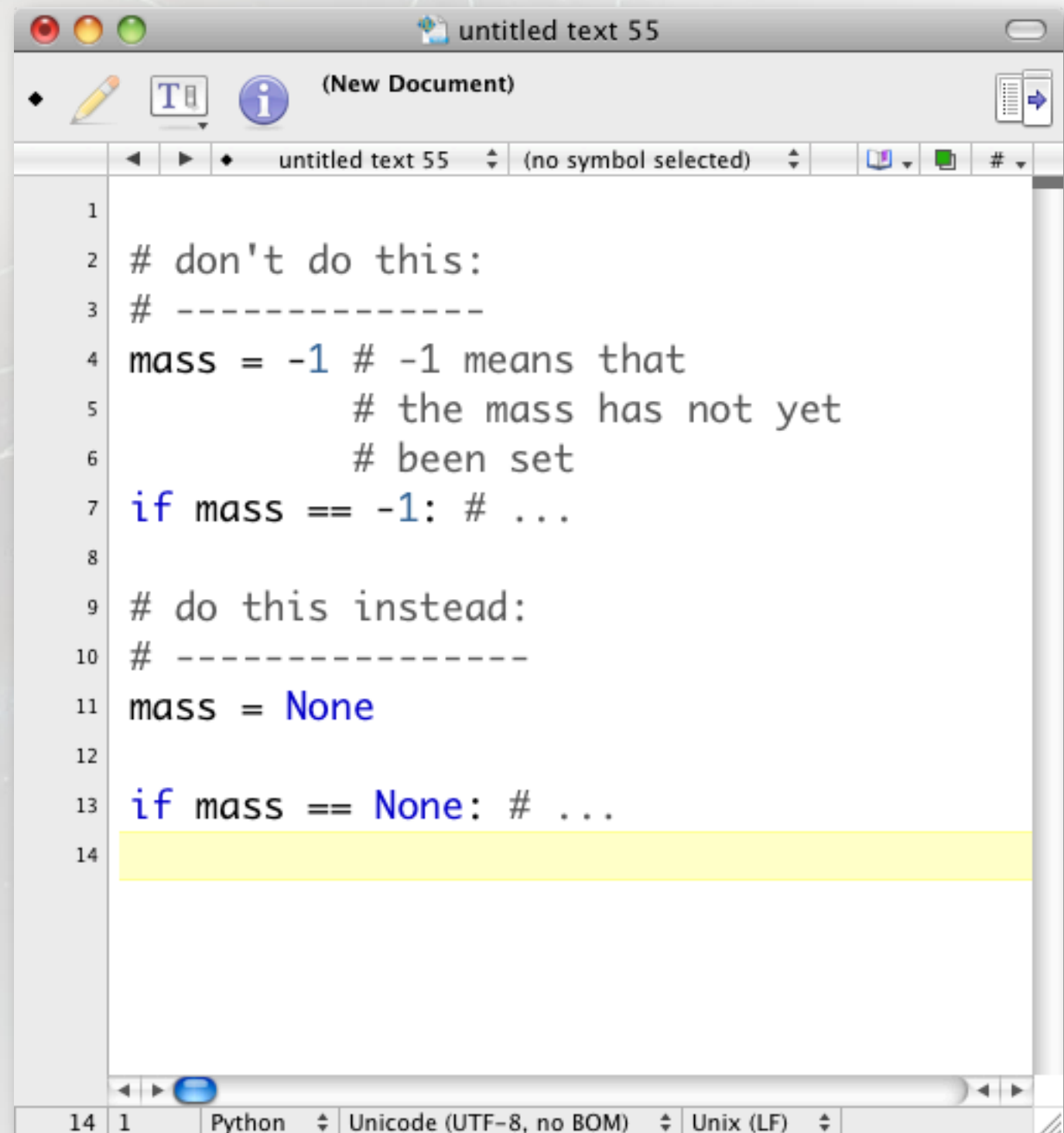
```
1
2 # this form
3 time = "It's five o'clock."
4
5 # is better than
6 time = 'It\'s five o\'clock.'
7
8 a = "Ray, when someone asks you \
9 if you're a god you say, 'Yes!'"
10
11 b = "Roads? Where we're going, " +
12     "we don't need roads."
13
14 c = "line 1" + "\n" + "line 2"
15
16 d = '''this is
17 all a single string
18 with the linefeeds included.'''
19
20 e = "col 1" + "\t" + "col 2"
21
```

newline

tab

# None

**None** is a special value that indicates null. Use this, for example, to indicate a variable has not yet been set or has no value rather than some number that has to be “interpreted”.



```
1
2 # don't do this:
3 # -----
4 mass = -1 # -1 means that
5           # the mass has not yet
6           # been set
7 if mass == -1: # ...
8
9 # do this instead:
10 # -----
11 mass = None
12
13 if mass == None: # ...
14
```



# Containers – Tuples and Lists

## Tuples

Groups of items

Can mix types

Can't be changed once created (immutable)

```
a = (1,2,3)
b = tuple() # empty tuple
c = ('a', 1, 3.0, None)
```

## Lists

a.k.a. arrays

Can mix types

Mutable

Lists, as proper OO objects, have built-in methods.

```
a = [5,3,6,True,5,5]
b = list() # new, empty list

# add items to a list
b.append(86)
b.append(99)

print len(b) # number of items in b

a.sort() # sort elements in place
a.reverse() # reverse elements in place
a.count(5) # number of times "5" appears in list

print a.sort() # returns "None"
print sorted(a) # does not modify a
print sorted(a, reverse=True) # reverse order
```

## Slices

```
a = ['a', 'b', 'c', 'd', 'e', 'f']
print a[3:5] # ['d', 'e'], 4th up to 5th item (not inclusive)
print a[-1] # f, i.e., last item
print a[:3] # ['a', 'b', 'c'], first 3 items
print a[2:] # ['c', 'd', 'e', 'f'], all items from 3rd to end
print a[:] # whole list
```

# Containers – Dictionaries

## Dictionaries

A group of items that are accessed by a value.

Arrays are accessed by index - the order is important. To access a given item, you have to know where it is or search for it.

A lot of data isn't inherently ordered. Takes ages of people in a family. You don't think "Bart was the third one born, so must be 10." You mentally map the name to the age.

`ages[key] = value`

dictionary  
name

can be almost any type - numbers,  
strings, objects (but not lists)

can be any type

**Dictionaries are not ordered.** You can iterate over them, but the items can be returned in any order (and it won't even be the same twice).

(Compare this idea to the everything box...)

Note: Called hashes or associative arrays in Perl, available as `std::map` in C++.

```
a = [100, 365, 1600, 24]

a[0] # first item
a[3] # 4th item

ages = dict()
ages['Lisa'] = 8
ages['Bart'] = 10
ages['Homer'] = 38

len(ages) # no of items in dictionary

ages.keys() # array of all keys
ages.values() # all values
del ages['Lisa'] # removes item
ages.has_key('Marge') # returns False
ages.clear() # removes all values

ages = {'Lisa':8, 'Bart':10, 'Homer':38}
```

shorthand method of creating a dictionary

# Control Structures

## *for* Loops

In C, we delineate blocks of code with braces – whitespace is unimportant (but good style).

```
void my_c_function {  
    // function code here  
}
```

In Python, the whitespace is the *only* way to delineate blocks (because it's good style).

```
for simpson in ages.keys():  
    print simpson + " is " + ages[simpson] + "years old."  
  
a = 12 # this is outside of the loop
```

You can use tabs or spaces to create the indentation, but you cannot mix the two. Decide which way you want to do it and stick to it. People debate which to use (and if you can be swayed, I *highly* recommend tabs).

### Example:

Given an array a of 10 values, print each value on a line.

C/C++

Python

```
# given an array of 10 values  
for (int i=0;i<10-1;i++) {  
    value = a[i]  
    printf ("%d", value)  
}  
  
for value in a:  
    print value
```

Can be anything in the list, and can create them on the fly:

```
for string in ['E','A','D','G','B','e']:  
    # do something
```



# Control Structures

If you *do* need an index in the loop:

```
a = ['a', 'b', 'c', 'd', 'e']
for index, item in enumerate(a):
    print index, item
```

```
# Output
# 0 a
# 1 b
# 2 c
# 3 d
# 4 e
```

*if* statement

```
if expression:
    statement 1
    statement 2
elif expression:
    pass
elif expression:
    ...
else
    statement 1
    statement n
```

expressions are  
boolean statements

*while* loop

```
# How many times is this
# number divisible by 2?
value = 82688
count = 0
while not (value % 2):
    count = count + 1
    value = value / 2
print value
print count
```

```
if True:
    # debug statements
```

useful for debugging; set  
to False when done



# Printing Variables

```
a = 12.4 # type is float (f)
b = 5 # type is integer (d = decimal)

print "The value of a is: %f" % a
print "The values of a is %f and the value of b is %d" % (a, b)

Change float output:
print "The value of a is: %.3f" % a # three decimal places
```

Note the need for parentheses with more than one value.

This is standard `printf` style formatting - google "printf format" for examples.

# Files

## Open a File

```
filename = "rc3_catalog.txt"  
f = open(filename)  
rc3_catalog_file = open(filename)
```

bad style - be  
descriptive in your  
variable names!

The actual filename is an input to your program. Try to abstract your inputs and place them at the top of the file.

Code defensively – what if the file isn't there? You'll be surprised how much time this will save you.

```
try:  
    rc3_catalog_file = open(filename)  
except IOError:  
    print "Error: file '%s' could not be opened." % filename  
    sys.exit(1)
```

- Minimize how much you put in the `try:` block.
- Determine what the error is by making the code fail in a simple program.

# Files

Read over all of the lines in the file:

```
for line in rc3_catalog_file:  
    if line[0] == "#":  
        continue  
    line.rstrip("\n")  
    values = line.split()
```

skip lines that begin with a '#'

strip the newline character from each line (split also removes \n)

separate the values by whitespace and return as an array

Write to another file:

```
output_file = open("output_file", 'w')  
output_file.write(a, b)
```

# try/except

```
import sys

a = 1
b = 0

print a / b

# Result:
# ZeroDivisionError: integer division or modulo by zero

try:
    c = a / b
except ZeroDivisionError:
    print "Hey, you can't divide by zero!"
    sys.exit(1) # exit with a value of 0 for no error, 1 for error
```

You don't have to exit from an error – use this construct to recover from errors and continue.

```
try:
    c = a / b
except ZeroDivisionError:
    c = 0

# continues
```

```
# check if a dictionary has
# a given key defined
try:
    d["host"]
except KeyError:
    # undefined, set default value
    d["host"] = localhost

# Although this command does the same thing!
d.get("host", "localhost")
```



# try/except

called only when  
`try` succeeds

provides the opportunity  
to clean up anything  
previously set up –  
always called

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print "division by zero!"
...     else:
...         print "result is", result
...     finally:
...         print "executing finally clause"
...
>>> divide(2, 1)
result is 2
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

(From the Python documentation.)

# Casting

Where appropriate, you can covert between types:

```
a = "1234" # this is a string
b = int(a) # convert to an integer

# but to be safer...

try:
    b = int(a)
except ValueError:
    b = None
```

Other examples:

```
a = '12.3e4'

print float(a) # 123000.0

print complex(a) # (123000+0j)

#print int(a) # ValueError

print int(float(a)) # 123000

print bool(a) # True

print str(complex(a)) # (123000+0j)
```

# Code Defensively – asserts

As your program runs, you make certain assumptions about your code. For example, we have an array that some process fills, and we assume it won't be empty.

If my\_values is empty, this loop is skipped silently.

```
my_values = list()
# some code to populate my_values

assert len(my_values) > 1, "my_values was empty!"
for i in my_values:
    # do stuff
```

If this fails, then the exception `AssertionError` is thrown and this message is printed out.

Be liberal with `assert` statements - they cost nothing. When your script is ready for production use, you can turn them off in two ways:

header in file

```
#!/usr/bin/python -O
```

command line

```
% python -O myScript.py
```

Can perform more than one check:

```
assert a > 10 and b < 20, "Values out of range."
```

# List Comprehension

Take the numbers 1-10 and create an array that contains the square of those values.

One of the nicest features of Python!

List comprehension generates a new array.

```
a = range(1,10+1)

a2 = list()
for x in a:
    a2.append(x**2)

a2 = [x**2 for x in a]
```



Using a for loop

Using list comprehension

Can also filter at the same time:

```
a = range(1,50+1)
# even numbers only
b = [x for x in a if x % 2 == 0]
```

Convert data types:

```
# read from a file
a = ['234', '345', '42', '73', '71']
a = [int(x) for x in a]
```

Call a function for each item in a list:

```
[myFunc(x) for x in a]
```

← can ignore  
return value



# Functions / Methods

document function  
with triple quotes

```
def myFormula(a, b, c, d):  
    ''' formula: (2a + b) / (c - d) '''  
    return (2*a + b) / (c - d)
```

indent as with loops

can set default values on  
some, all, or no parameters

```
def myFormula(a=1, b=2, c=3, d=4):  
    ''' formula: (2a + b) / (c - d) '''  
    return (2*a + b) / (c - d)  
  
print myFormula(b=12, d=4, c=5)
```

Note order doesn't matter when  
using the names (preferred method).

If a default value is set, you don't have to call it at all.

Useful math tools:

```
import math  
  
# constants  
a = math.pi  
b = math.e  
  
c = float("+inf")  
d = float("-inf")  
e = float("inf")  
f = float("nan") # not a number  
  
def myFormula(a, b, c, d):  
    ''' formula: (2a + b) / (c - d) '''  
    num = 2 * a + b  
    den = c - d  
    try:  
        return num/den  
    except ZeroDivisionError:  
        return float('inf')  
  
# tests  
math.isnan(a)  
math.isinf(a)
```

# Functions / Methods

## Passing parameters into function / methods.

Unlike C/C++, the parameter list is dynamic, i.e. you don't have to know what it will be when you write the code.

You can also require that all parameters be specified by keywords (kwargs).

Note two **\*\*** here vs. one above.

```
def myFunction2(**kwargs):    kwargs = keyword arguments
    for key in kwargs.keys():
        print "Value for key '%s': %s" % (key, kwargs[key])

myFunction2(name='Zaphod', heads=2, arms=3, president=True)

# Output:
# Value for key 'president': True
# Value for key 'heads': 2
# Value for key 'name': Zaphod
# Value for key 'arms': 3
```

Note the output order is not the same (since it's a dictionary).

Accepts any number of arguments (of any type!)

```
def myFunction(*args):
    for index, arg in enumerate(args):
        print "This is argument %d: %s" % (index+1, str(args[index]))

myFunction('a', None, True)

# Output:
# This is argument 1: a
# This is argument 2: None
# This is argument 3: True
```

Can be mixed:

```
def myFunction3(*args, **kwargs):
    print "ok"

myFunction3()
myFunction3(1, 2, name="Zaphod")
myFunction3(name="Zaphod")
myFunction3(name="Zaphod", 1, True)
```

zero args are ok

Invalid - named arguments must follow non-named arguments (as defined).

# Odds and Ends

## Range

```
range(10)      # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
range(10,20)   # [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
range(10,20,2) # [10, 12, 14, 16, 18]
```

useful in loops

(start, stop, step) - step can only be an integer

```
[x * 0.1 for x in range(0, 10)]
```

generating ranges in non-integer steps

## Objects and Copies

Does not make a copy – these are the same objects!

Copies all of the items into a new object.

```
ages = {'Lisa':8, 'Bart':10, 'Homer':38}
simpsons = ages
ages['Bart'] = 9
print simpsons['Bart'] # output: 9
```

```
ages = {'Lisa':8, 'Bart':10, 'Homer':38}
simpsons = ages.copy()
ages['Bart'] = 9
print simpsons['Bart'] # output: 10
```



# Odds and Ends

The *in* operator:

```
a = ['a', 'b', 'c', 'd', 'e', 'f']  
print 'a' in a # True  
print 'x' not in a # True
```

Create Strings from Lists  
with a Delimiter

```
strings = ['E', 'A', 'D', 'G', 'B', 'e']  
print "|".join(strings)  
# Output: E|A|D|G|B|e
```

## Operator Overloading

We know '+' adds two numbers, but it also "adds" two strings together. We can define that operator to mean custom things to our own objects.

(This is a powerful feature!)

added a new  
init method  
that takes a  
radius

override +  
operator

```
class Circle(Shape):  
    radius = 0.0  
  
    def __init__(self, r=0.0):  
        self.radius = r  
  
    def area(self):  
        return math.pi * self.radius * self.radius  
  
    def __add__(self, other):  
        c = Circle()  
        c.radius = self.radius + other.radius  
        return c  
  
c1 = Circle(r=5)  
c2 = Circle(r=10)  
c3 = c1 + c2  
  
print c3.radius # Result: 15
```

radius is optional

now we can add two Circle objects together to create a new Circle



# Further Reading

This is a great reference for Python. Keep this bookmark handy.

<http://rgruet.free.fr/PQR26/PQR2.6.html>

Several people have emailed me this – it's also a good introduction.

<http://openbookproject.net//thinkCSpy/>

This web page has over one hundred “hidden” or less commonly known features or tricks. It's worth reviewing this page at some point. Many will be beyond what you need and be CS esoteric, but lots are useful. StackOverflow is also a great web site for specific programming questions.

<http://stackoverflow.com/questions/101268/hidden-features-of-python>

And, of course, the official Python documentation:

<http://docs.python.org>

Finally, if you are not familiar with how computers store numbers, this is mandatory reading:

<http://docs.python.org/tutorial/floatingpoint.html>