

# Starting with Python

# Why Python?

- 1) readable, compact, simple syntax
- 2) documented
- 3) memory managed
- 4) types inferred at run-time
- 5) dynamic: classes are objects
- 6) batteries included:  
extensive software library
- 7) designed avoiding “feature-creep”
- 8) only hundred times slower than C\*
- 9) high-level: useful data types  
dictionary, sets, sequences, lists, complex, ...
- 10) Allows multiple styles of programming

→ **effective scripting language**



# Running simple Scripts

```
>python your_file.py
```

```
>ls
```

```
-> your_file.py      your_file.pyc
```

**Compiled to .pyc file**

# Basic syntax

- No semicolon;
- C-based syntax, less brackets
- Control flow via indentation

```
if danger :
```

```
    if smaller_than_you() :
```

```
        fight()
```

```
    else:
```

```
        run()
```

# Basic Datatypes

Type	Example	Convert
boolean	True	bool()
integer	100000000000000019	int()
floating point	3.14	float()
complex	1+2j	complex()
string	'singly quoted string'	str()
string	"double quoted string"	str()
tuple	(1,2,'abc')	tuple()
list	[1,2,'abc']	list()
dictionary	{ "tel": 4396, 'name' : 'Python', 1 : 2 }	dict()

## Variable Assignment

```
a = 2.1
```

```
b = 'some string'
```

```
c = [ 'list', 'of', 'strings' ]
```

# Print (3.1)

## Using **print** to displaying values

```
a = 3
```

```
b = 'abc'
```

```
c = [ 1, 2, 'muchos' ]
```

```
print(a,b,c)
```

```
3 abc [ 1, 2, 'muchos' ]
```

```
# use # to write comments
```

```
print(a+2, a**2 < 10, max(a**3, 10))
```

```
# Python output shown using green text
```

```
5 True 27
```

# Scope

- Declarations are global
- Cannot be modified in local scope

```
c = 1
def f(n):
    print c + n
def g(n):
    c = c + n
```

```
f(1) => 2
```

```
g(1) => UnboundLocalError: local variable
'c' referenced before assignment
```

# **CONTROL STRUCTURES**

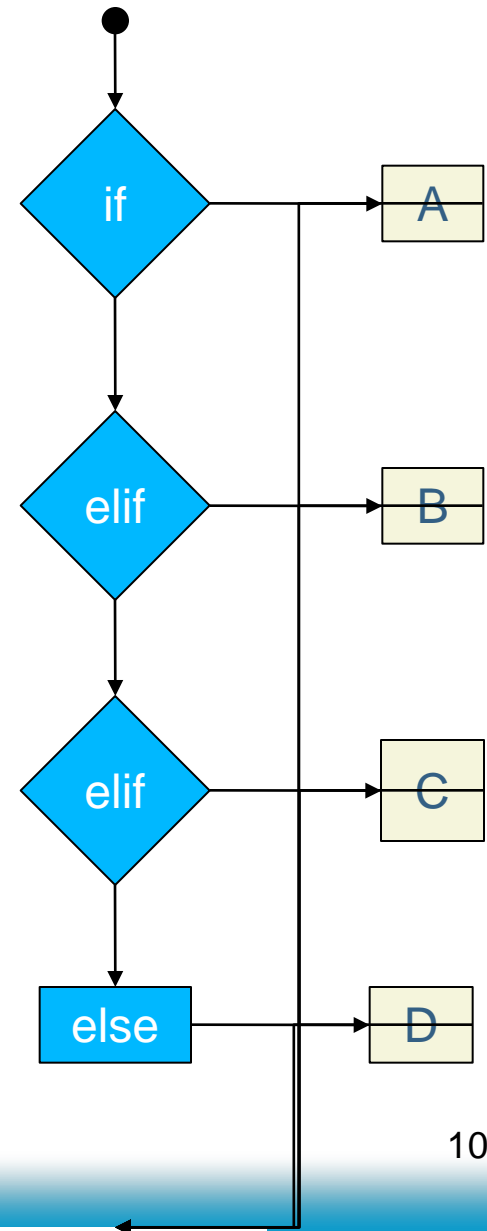
# if ...

```
# exactly one
if age <= 1:
    return age*Y1

# zero to n
elif age <= 2:
    return Y1 + (age-1)*Y2

# zero to n
elif COND:
    :
    .

# optionally
else:
    return Y1 + Y2+ (age-2)*YR
```



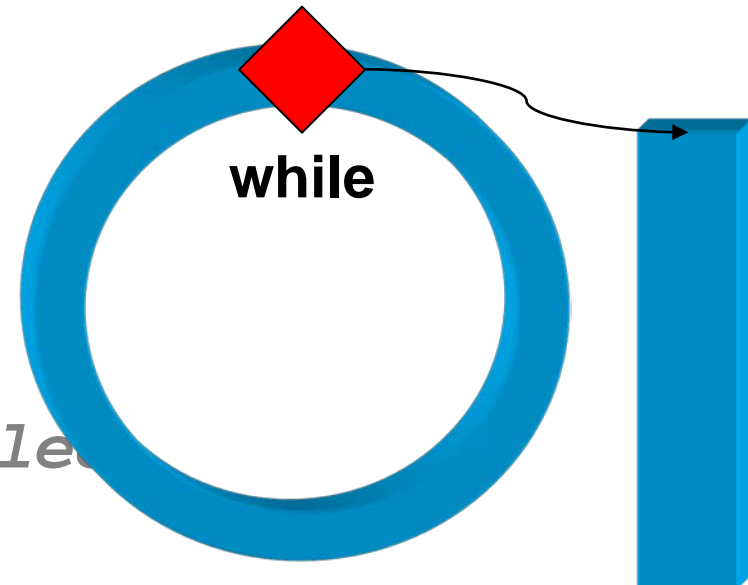
# while Loop

*# repeats command block  
# as long some condition is fulfilled*

```
while weather_good():  
    call_friend()  
    go_swim()  
    eat_icecream()
```

*# if condition never fulfilled*

```
else:  
    stay_home()
```



# for $x$ in $S$ : something( $x$ )

does something for each item  $x$  of  $S$

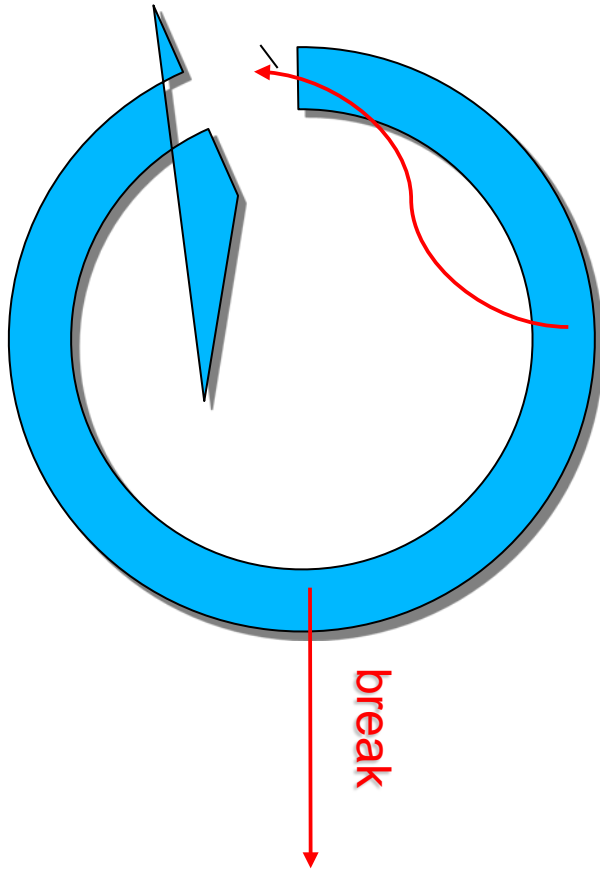
```
# archetypal for-loop  
# prints 0,1, ..., n-1  
for i in range(0,n):  
    print(p)
```



```
# prints ASCII code for each letter in string ABC  
ABC = 'abcdefghijklmnopqrstuvwxyz'  
for letter in ABC:  
    print(ord(letter))
```

```
# hunts for every animal in list zoo  
zoo = [ 'egg', 'dog', 'duck', 'elephant' ]  
for animal in zoo:  
    safari_hunt(animal)
```

# Control statements



**break**

escape from enclosing loop

**continue**

begin next iteration

**pass**

Do nothing (indentation!)

# for vs while

**for** usually compacter, **while** more general.

**for** - doing *X* for each element of *S*.


- can simulate **while** loops (using iterators)

**while** – repeating *Y* as long *C* fulfilled.

- waiting for something to change.
- can easily simulate **for**-loops

```
# basic iterating over a sequence seq  
for x in seq:  
    do_something(x)
```

```
cnt = 0  
while cnt < len(seq):  
    x = seq[cnt]  
    do_something(x)  
    cnt += 1
```



# **FUNCTIONS**

# Function Definition

```
def function_name (arg1, arg2, ...):
```

```
    ''' This function does this and that.  
    (the 'doc_string').
```

```
    '''
```

*statements*

*indented*

*by*

*spaces*

```
def h2d_years(age, breed):  
    Y1 = pow(2, 3.6438)  
    return age*Y1
```

# Function Examples

- Defining

```
def cube(x):  
    return x**3.0
```

```
def alfify(x, y, alf=1):  
    return x**(-alf) +  
           y**alf
```

```
def hello(y):  
    print('Hello'+y)
```

- Calling

```
a = 2
```

```
b = 3
```

```
cube(b) → 27
```

```
alfify(a,b) → 3.5
```

```
min(a+b, a-b, -2) → -2
```

```
sum(a, b, 10) → 15
```

```
a < b → True
```

```
a==b → False
```

```
callable(a+b) → False
```

# Nesting Functions

- You may write
- ```
def outer(a,b,c):  
    x = a+b+c
```
- ```
    def inner(u, v):  
        if u>v:  
            return u+v  
        else:  
            return u*v
```
- ```
    return inner(x, b) + inner(x,c)
```
- → access everything outer can
- → simplify expressions in complex loops
- → not visible outside of outer
- → small functions passed around (and returned)



# Keyword arguments

Default argument values

```
def alfify(x,y,alf):  
    return x**-alf +  
           y**alf
```

Parameters

```
def alfify(x = 1,y =  
           2,alf = 1):  
    return x**-alf +  
           y**alf  
  
alfify()  
alfify(y=3)
```

# lambda

- Functional programming element
- Anonymous function

```
d = lambda(x: x+2)
```

Equivalent to

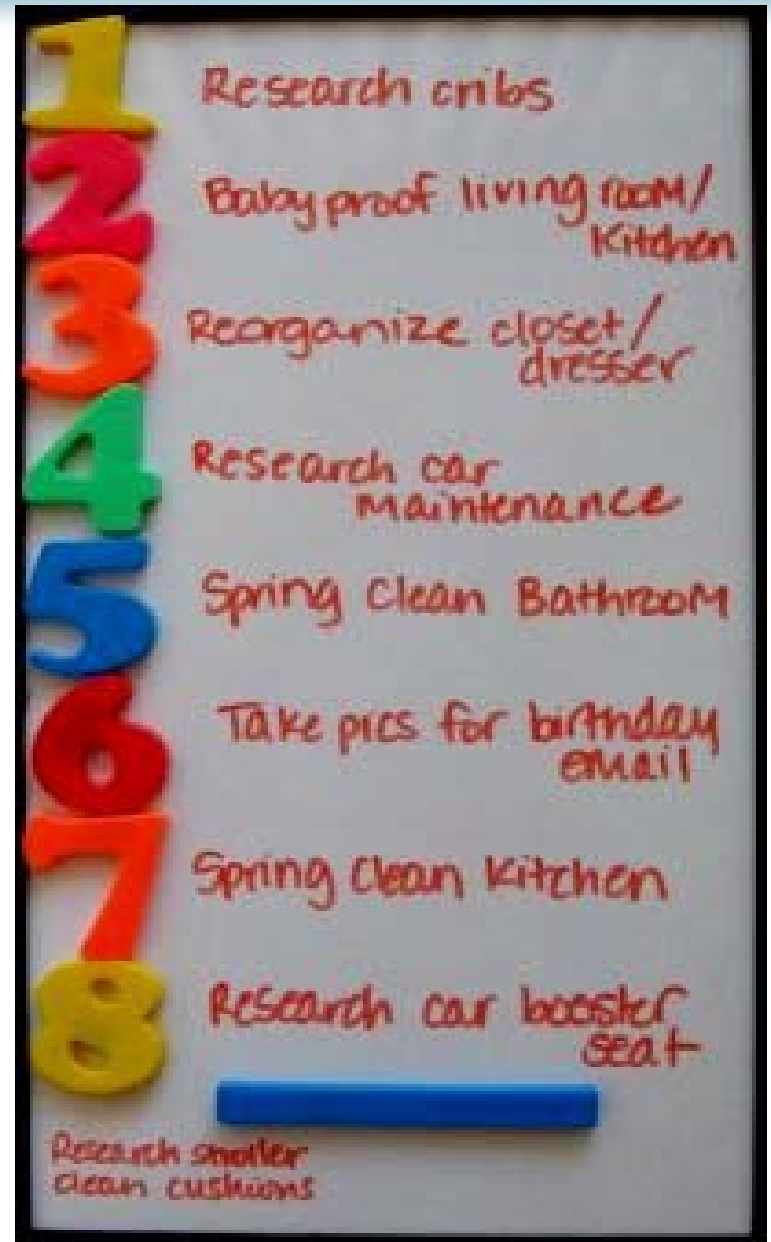
```
def d(x):  
    return x+2
```

# docstrings

```
def h2d_years(age, breed='mix'):  
    '''Converts human years into dog years.  
    'breed' maybe None.  
    ...  
    [...]
```

- ♪ first thing in function body
- ♪ string surrounded by `'''triple quotes'''`  
(may include new lines, apostrophes `'` and quotation marks `"`)
- ♪ used to generate online documentation  
(`help` / `pydoc`)

# Lists



# Creating Lists

```
# empty list  
l = []
```

```
# heterogeneous: mix strings, numbers,  
# functions, modules, objects  
l = [ 'spam', 'eggs', 100, max ]
```

```
# integers between x and y, step s  
l = range(x,y,s)
```

```
# attempts to make a list out of x  
l = list(x)
```

```
# nesting allowed  
l = [ range(0,101), list('abc'), [[]] ]
```

# Working with lists

```
aa = [ 'spam', 'eggs', 100, 1234, max, min ]
```

## □ *mutable*

```
aa[2] += 2*aa[3]
```

```
aa → [ 'spam', 'eggs', 2968, 1234, max, min ]
```

## □ + and \* work like strings

```
[1, 2, 3]*3 → [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

```
[1, 2]+[5, 6] → [ 1, 2, 5, 6 ]
```

## □ Nested lists allowed

```
[ 1, 2, 3, [ 'q', 'p', [ 'alpha' ] ] ]
```

## □ can be “sliced”...



# List Operation

```
seq = [0,1]
```

```
l.append(x)
```

Appends at the end of the list.

```
seq.append(2) → [0, 1, 2]
```

```
l.extend(t)
```

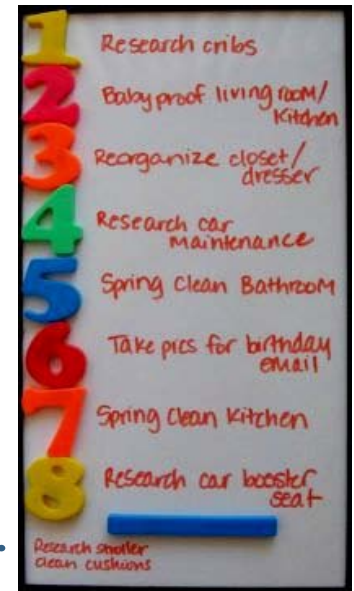
Append the items of *t* to the end of the list.

```
seq.extend( (3,4) ) → [0, 1, 2, 3, 4]
```

```
l.insert(i,x)
```

inserts *x* at position *i*, shifting remaining items right.

```
seq.insert(2, 10) → [0, 1, 10, 2, 3, 4]
```



# List Operation

`l = [3, 2, 4, 1, 0]`

## **`l.insert(i,x)`**

inserts `x` at position `i`, shifting remaining items right.

`seq.insert(2, 10) → [0, 1, 10, 2, 3, 4]`

## **`l.sort()`**

Sorts `l`, in place.

`l.sort() → [0, 1, 2, 3, 4]`

## **`l.reverse()`**

Reverses `l`, in place.

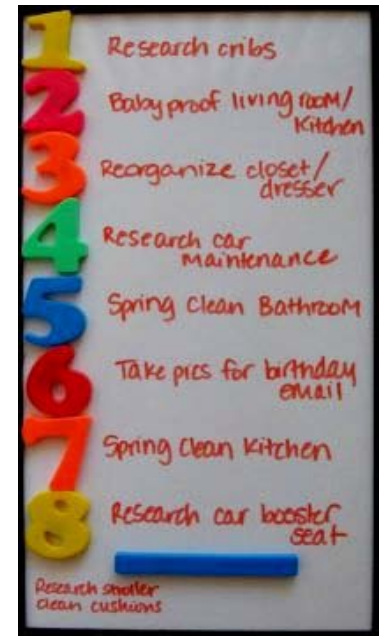
`l.reverse() → [4, 3, 2, 1, 0]`

## **`l.remove(x)`**

Removes first item equal to `x` from list `l`.

`seq.remove(10) → [0, 1, 2, 3, 4]`

`>>>help(list)`



# Slicing

**`seq[start:end:stride]`**

```
seq = range(100) #< numbers between 0 to 99
```

□ **22<sup>th</sup> item**

```
seq[22] → 22
```

□ **first 33 items**

```
seq[:33] → [ 0, 2, ..., 32 ]
```

□ **last 33 items**

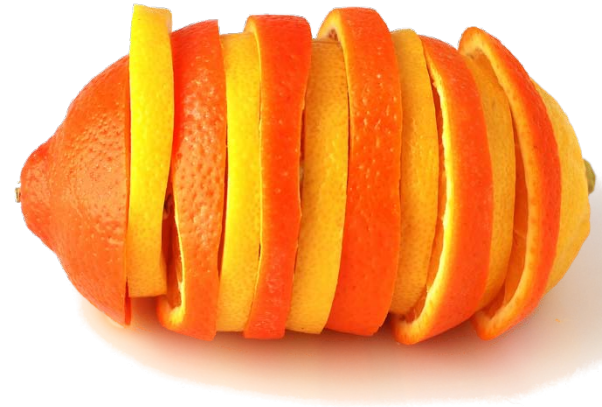
```
seq[-33:] → [ 67, 68, ..., 98, 99 ]
```

□ **every 2<sup>nd</sup> item**

```
seq[::2] → [ 0, 2, ..., 96, 98 ]
```

□ **Creating a copy**

```
tir = seq[:]
```



# Slicing for Lists

**seq[start:end:stride]**

```
# Example: numbers between 0 to 99  
seq = range(100)
```

- assignment supported

```
seq[::3] = [-1] * 34
```

- length can change

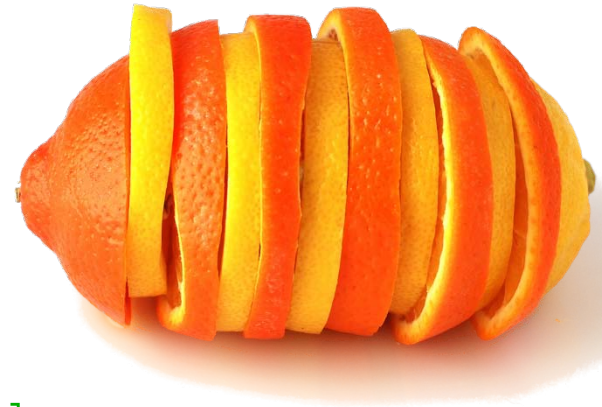
```
seq[2:98] = [ "and so on" ]  
seq → [ 0, 1, "and so on", 98, 99 ]
```

- slice can be deleted

```
del seq[::2]  
seq → [ 1, 3, 5, ..., 95, 97, 99 ]
```

works only if *seq* is mutable!

lists: yes, strings: no.



# Stacks and Queues

## First in, first out

```
stack = [4,5,6]
```

```
stack.append(7)
```

```
[4,5,6,7]
```

```
stack.pop()
```

```
[7]
```

## Access from both ends

```
from collections import  
deque
```

```
queue = deque(["Eric",  
"John", "Michael"])
```

```
queue.append("Terry")
```

```
queue.popleft()
```

```
'Eric'
```

```
queue.pop()
```

```
"Terry"
```

# Sequence Types

## List has immutable twin called *tuple*!

- Behaves identically except but can not **del** or reassign items.
- Creating tuples like lists:  
t = () #< empty tuple  
t = (1,) #< tuple with only 1 item  
t = (1, 'abc', 3)  
t = (1, 'abc', 3,)  
t = tuple(x) #< converts x into a tuple
- Converting (immutable) tuple into (mutable) list  
my\_list = list(my\_tuple)

- strings, tuples and lists are

*Sequence Types*

10/19/2010



# Working w/ Sequence Types

string, unicode string, list, tuple, xrange

**$x$  in  $s$**

True, if item of  $s$ , equal to  $x$

**$x$  not in  $s$**

False if item of  $s$  equal to  $x$

$s + t$

concatenation  $s$  and  $t$

$s * n$

$s$  concatenated  $n$  times

$s[i]$

$i$ th item of  $s$ , origin 0

$s[i:j]$

slice of  $s$  from  $i$  to  $j$

$s[i:j:k]$

slice of  $s$  from  $i$  to  $j$ , step  $k$

$\text{len}(s)$

length of  $s$

$\text{min}(s)$

smallest item of  $s$

$\text{max}(s)$

biggest item of  $s$

$\text{any}(s)$

True, if at least one item logically true

$\text{all}(s)$

True, if all items in  $s$  logically true



# Sequence: applying **in**

*# simplified quest for happiness*

luv = 'L.O.V.E'

air = 'hay hay hay hay hay L.O.V.E' + 33\*'hay'

**if** luv **in** air:

    sing()

*# readable logical expressions:*

**import** keyword

**if** word **in** keyword.kwlist:

**print** "Error: '", word, "' is a reserved keyword!"



# Modules

## Modules bundle functionality

(examples: handling time and date, interface to O.S.)

```
import sys
```

```
print(sys.path)
```

```
print(sys.argv)
```

Name of module

*path* is attribute\* of *sys*  
(search path for modules)

attribute with command line arguments  
(for scripts)

# Import statement

```
import quite_long.xyz as qux
```

- 1) Searches in *sys.path* for **quite\_long/xyz.py**
- 2) Creates module object
- 3) Assigns **qux** from module object

Names of modules not special: you *can* write

```
sys = 1  
os = 'abc'
```

# Attributes

```
def human2dog_years(age, breed):  
    """Converts human years into dog years.  
    'breed' maybe be None.  
    """  
    [...]  
x = human2dog_years  
print x.__name__
```



- ingredient of nearly every pythonic thing  
(including functions, objects, classes, modules, container)
- if **thing** has attribute **'abc'**  
→ access by writing **thing.abc**
- gives access to functionality  
provided by *thing*



# String Attributes

```
my_string = ' My_Example_String '
```

```
my_string.startswith('My') → True
```

```
my_string.endswith('ing') → True
```

```
nu = my_string.replace('_ ', ' ')  
→ ' My Example String '
```

```
my_string.strip() → 'My Example String'
```

## Capitalization

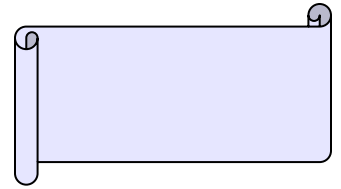
```
nu.upper() → 'MY EXAMPLE STRING'
```

```
nu.lower() → 'my example string'
```

```
□ '~'.join( [1,2,3] ) → 1~2~3
```

```
□ there is more....
```

```
>>>help(str)
```



# Conversion to/from Strings



From strings

**float(*s*)** → converts string *s* into floating point (double precision)

**int(*s*)** → converts string *s* into integer

**bool(*s*)** → converts string *s* into True or False

Into strings

**str(*x*)** → convert *x* into string (aka toString())

**repr(*x*)** → *x* into Python expression

# String-Operation



```
a = 'pirfyx'
```

```
b = "tail"
```

Concatenation:

```
a+b+"ing" → 'pirfyxtailing'
```

Repetition: repeat *b* three times:

```
b*3 → 'tailtailtail'
```

Index starts with 0:

```
a[2] → 'r'
```

Length of string:

```
len(a) → 6
```

```
len(b) → 4
```

# Formating strings

```
help(str.format())
```

```
i = 7
```

```
s = "dwarves"
```

```
"{} {}".format(i, s)
```

```
"7 dwarves"
```

```
"{: .3f}".format(9/7)
```

```
1.286
```

# Sets and frozen sets

Set: unordered collection with no duplicates

```
S = set(1, 2, 3, 1, 2, 3, 2, 4)
```

```
print(S)
```

```
{1, 2, 3, 4}
```

Frozenset: immutable

Union, intersection, difference available as  
overloaded operators

# Dictionary

- Associative arrays

```
tel = { 'krause': 1153,  
       'vingron': 1150 }
```

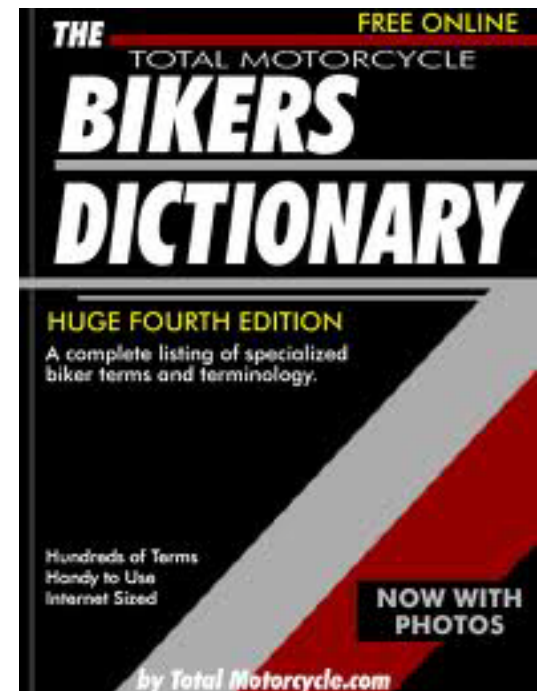
```
tel[ "krause" ]
```

1153

```
Del tel[ "krause" ]
```

```
Tel
```

```
{ "vingron" : 1150 }
```



# Looping through a dictionary

```
for key, value in tel.items():  
    print(key, value)
```

Better:

```
print(" ".join(map(str,  
tel.items())))
```

# Everything is an Object !

Functions are objects,  
Strings are objects,  
Lists, classes, errors, modules too.  
**Everything is an object!**

- (1) Good: every object can be ...
- (2) assigned to variables
- (3) passed around as arguments to
- (4) Stored in a list
- (5) Etc...



# Python Objects, Example

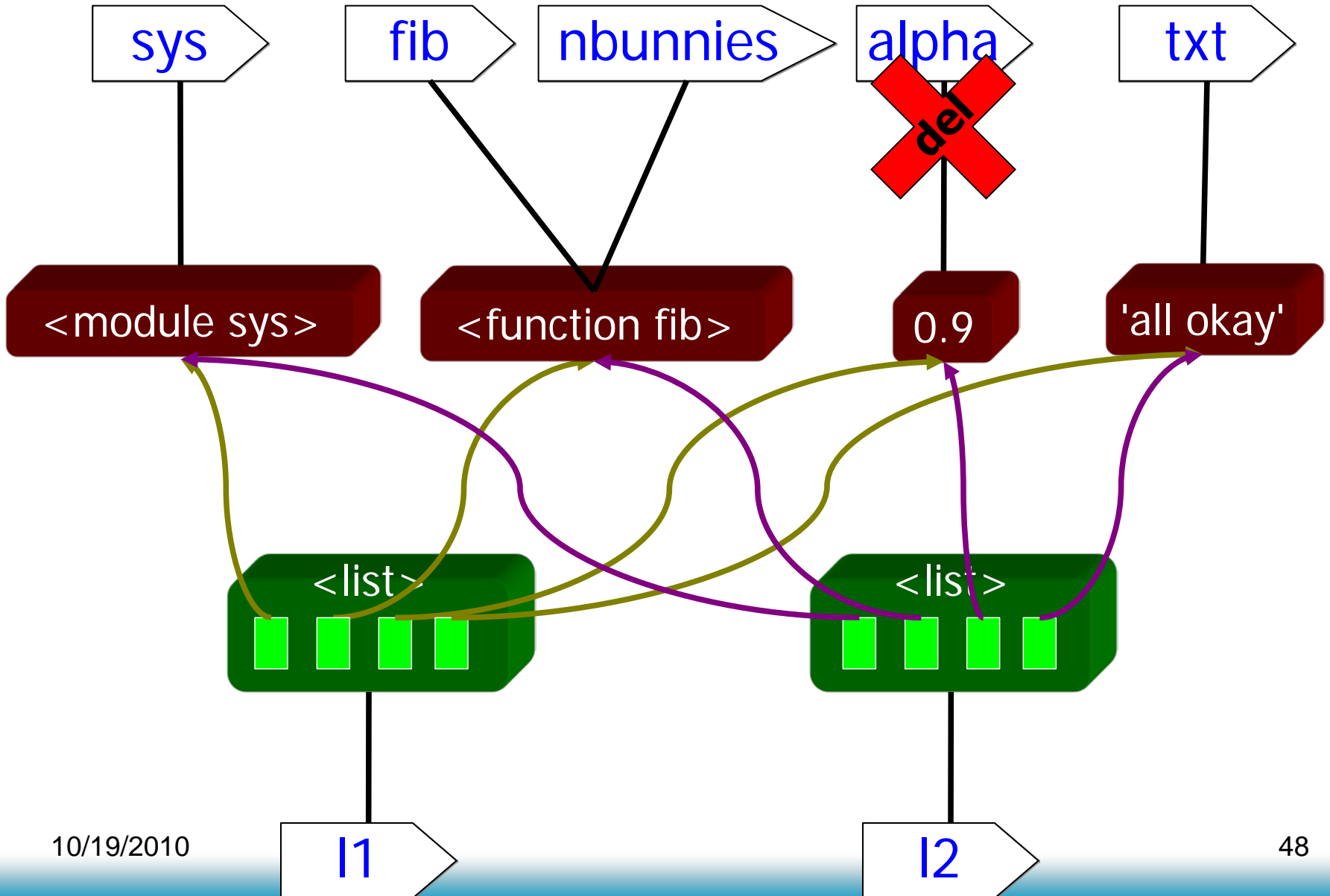
```
import sys

def fib(n, a=0, b=1):
    for x in range(n):
        a,b = b,b+a
    return a

alpha = 0.9
nbunnies = fib
txt = 'all okay'

l1 = [ sys, nbunnies, alpha, txt ]
l2 = [ sys, fib, alpha, txt ]
# remove alpha
del(alpha)
```

# Python and Objects



# Python Objects

## 1) Identity

→ unique and constant

→ **is** operator

## 2) (Data)Type

→ Define behavior

→ constant

→ obtained by `type()`

## 3) Value

→ can change unless immutable

→ accessed using attributes

10/19/2010



```
import sys
import os
```

```
abc = sys
sys = os
print sys is abc
print type(sys), type(abc)
```

# dogyears.py

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

'''dogyear demo script for converting,
   man years in dog years.

   It is a dogs life after all.
'''
import sys

# defining constants used in conversion
# dog aging in ...
Y1 = 12.5 # first human year
Y2 = 5.5  # second human year
YR = 4.5  # every year thereafter

def human2dog_years(age):
    """Converts human years into dog
    years.

    It is assumed that 'age' >= 0.
    """
    assert age <= 0, "too young"

    # between 0 and 1 human years old
    if age <= 1:
        return age*Y1
        # human2dog_years continued
        # between 1 and 2 human years old
    elif age <= 2:
        return Y1 + (age-1)*Y2

    # at least 2 human years old
    else:
        return Y1 + Y2+ (age-2)*YR

# digest command line arguments
if len(sys.argv) == 2:
    age = float(sys.argv[1])
    print "You are %.2f dog years old." \
        % human2dog_years(age)
else:
    print ""dogyears expects EXACTLY
    one argument.""
```

# Packing and Unpacking

**Packing:** assigning multiple variables to a tuple

```
t = a,b,c,d,e
```

**Unpacking:** assigning a sequence to multiple variables  
(aka multiple assignment)

```
a,b,c,d,e = t
```

```
q,d,c,s = '?::;'
```

```
u,n,i,x = range(1,5)
```

**Swap!**

```
u,n,i,x = n,u,x,i
```

# Regular expressions

```
import re
re.findall(r'\b[a-z]*', 'which
foot or hand fell fastest')
['foot', 'fell', 'fastest']
re.sub(r'(\b[a-z]+) \1', r'\1',
'cat in the the hat') 'cat in
the hat'
'tea for too'.replace('too',
'two') 'tea for two'
```

# Getting Help

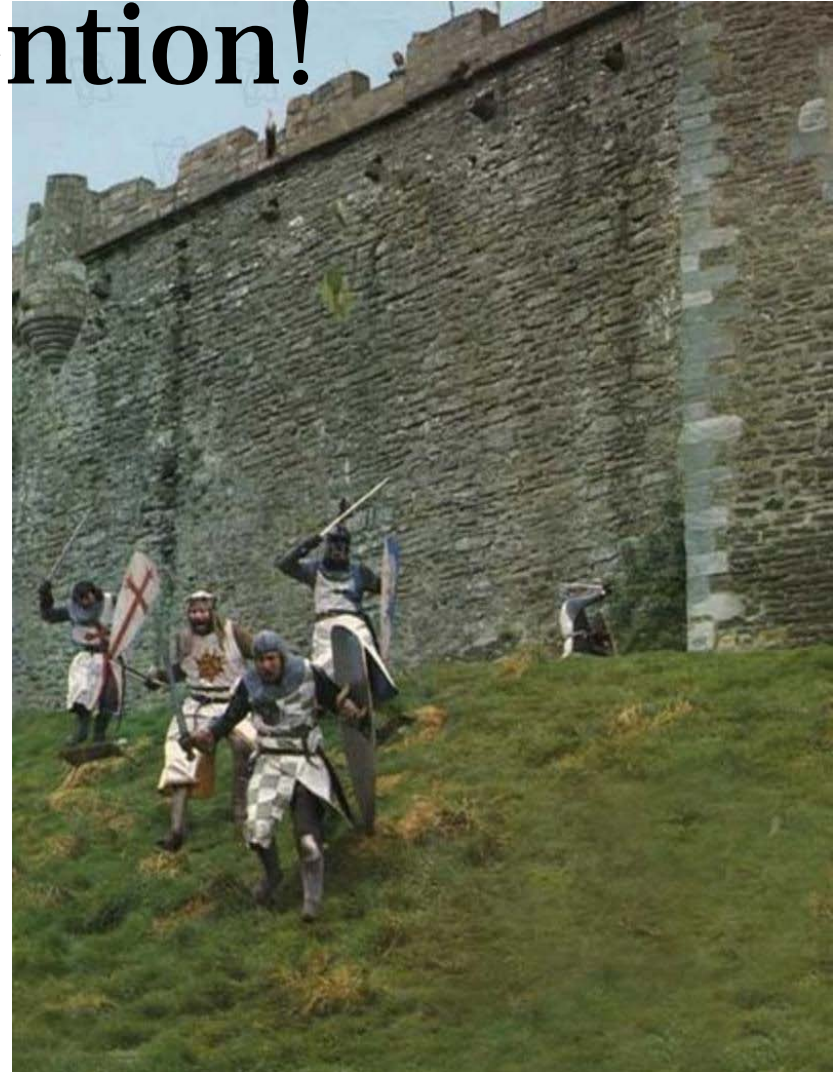
□ builtin help system:

\$ **pydoc *thing*** (command line help)

>>> **help(*thing*)** (builtin python help)

□ <http://docs.python.org>

Thank you for  
your attention!



# Example: Plotting with Python

```
import numpy as np
import matplotlib.cm as cm
from matplotlib.pyplot import figure, show,

# force square figure and square axes
fig = figure(figsize=(8,8))
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8], polar)

N = 20
theta = np.arange(0.0, 2*np.pi, 2*np.pi/N)
radii = 10*np.random.rand(N)
width = np.pi/4*np.random.rand(N)
bars = ax.bar(theta, radii, width=width, bottom=0.0)
for r,bar in zip(radii, bars):
    bar.set_facecolor( cm.jet(r/10.))
    bar.set_alpha(0.5)
show()
```

