

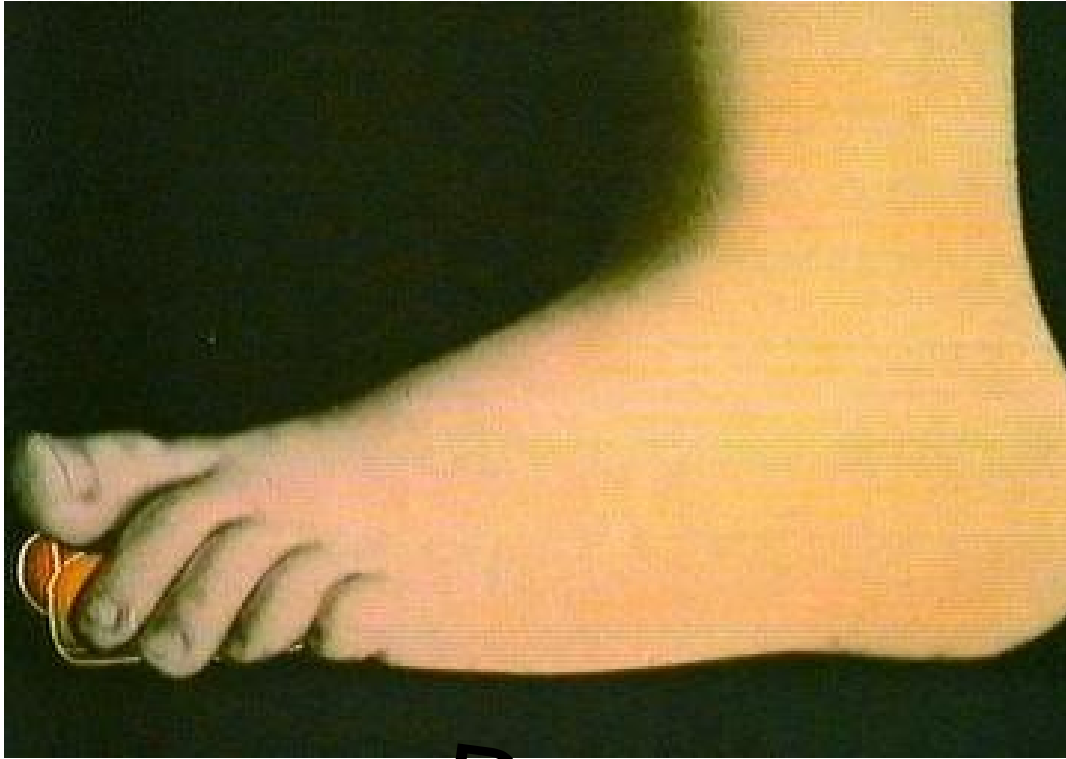
# `_init_` python(self): Introduction for Programmers

Rahul Mahale

<http://rahulmahale.wordpress.com/>

# Game Plan

- Lab I
  - Basic Introduction to python
  - Data types and its use.
- Lab II
  - Flow Control in Python
  - Methods(Functions)
- Lab III
  - Modules ,Classes
  - File Operations
  - Puzzles



# Python: Introduction for Programmers

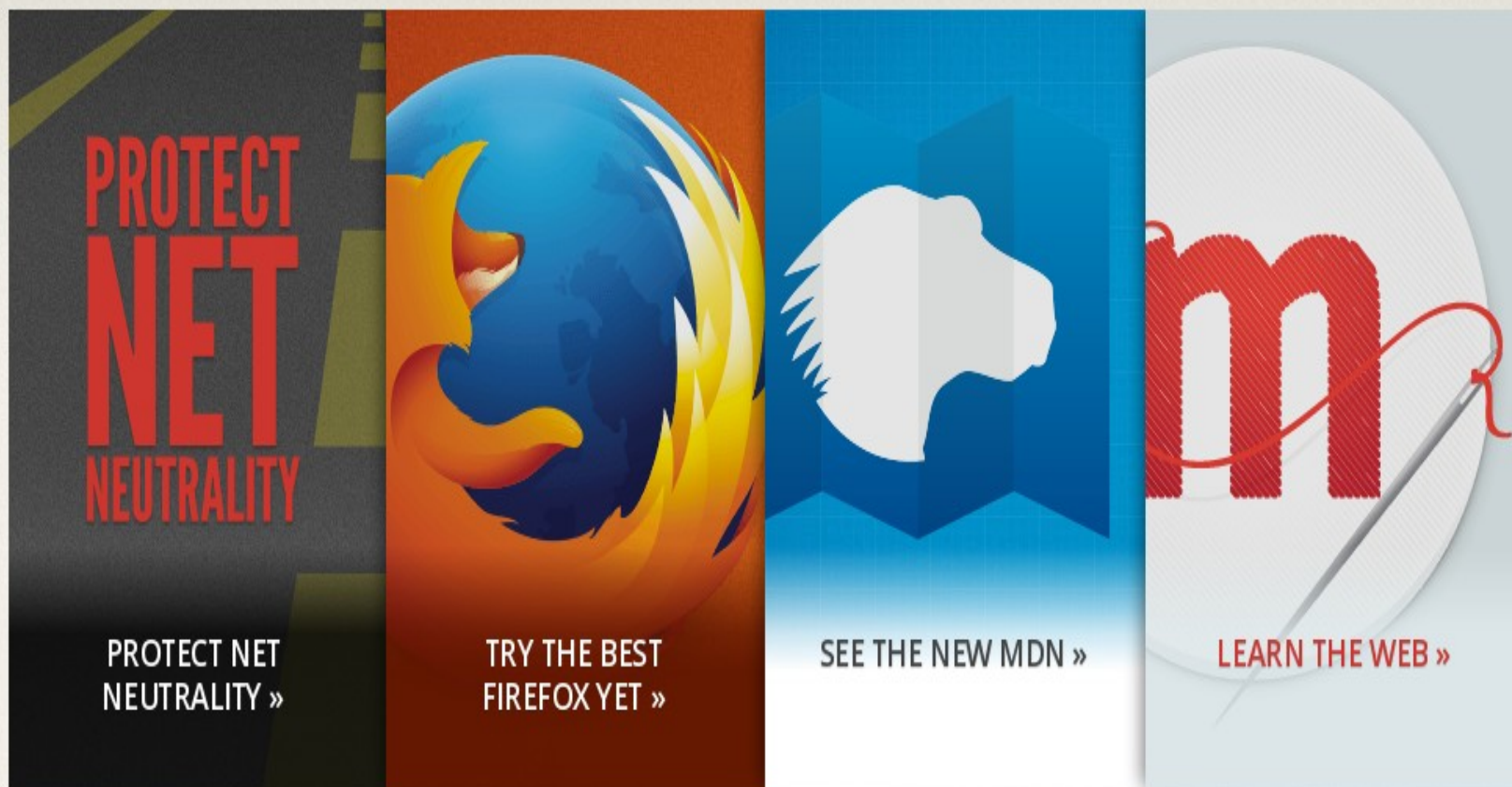
Python is good for ?

MISSION ABOUT PRODUCTS GET INVOLVED

mozilla ▾

# We are mozilla

Doing good is part of our code




Ac... 1 15:03 28 °C 9.06 KB/s

Instagram - Mozilla Firefox

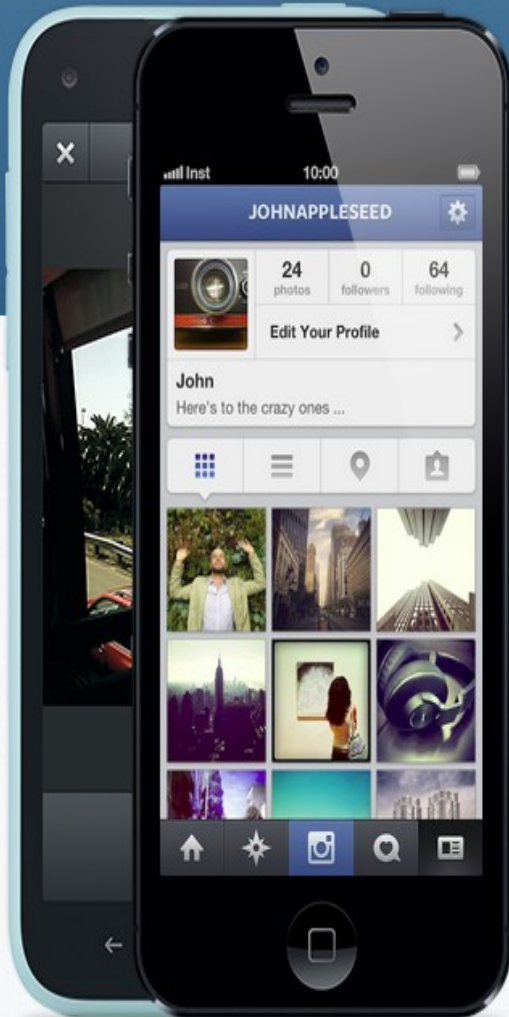
dding ... Interac... Impres... WordP... Integra... How T... cubew... Examl... WordP... Tools < ... m Home ... Instagram x

instagram.com/# Google



# Instagram

[Log in](#)





## Capture and Share the World's Moments

Instagram is a **fast, beautiful** and **fun** way to share your life with friends and family.

Take a picture or video, choose a filter to transform its look and feel, then post to Instagram — it's that easy. You can even share to Facebook, Twitter, Tumblr and more. It's a new way to see the world.

Oh yeah, did we mention it's free?

 Download on the  
**App Store**

 GET IT ON  
**Google play**



A semi-transparent overlay on the Pinterest homepage. The background of the overlay shows a person's silhouette looking at a cityscape. The text and buttons are centered.

**Pinterest**

# They used Pinterest to plan a dream trip

Join Pinterest to find (and save!) all the things that inspire you.

 **Continue with Facebook**

 **Sign up with Email**

Already have an account? [Log in now](#)

**45 seconds**  
to sign up (free!)

**25+ Billion**  
Pins to explore

# Science



Collaborative drug  
discovery



Glue language for many  
platforms



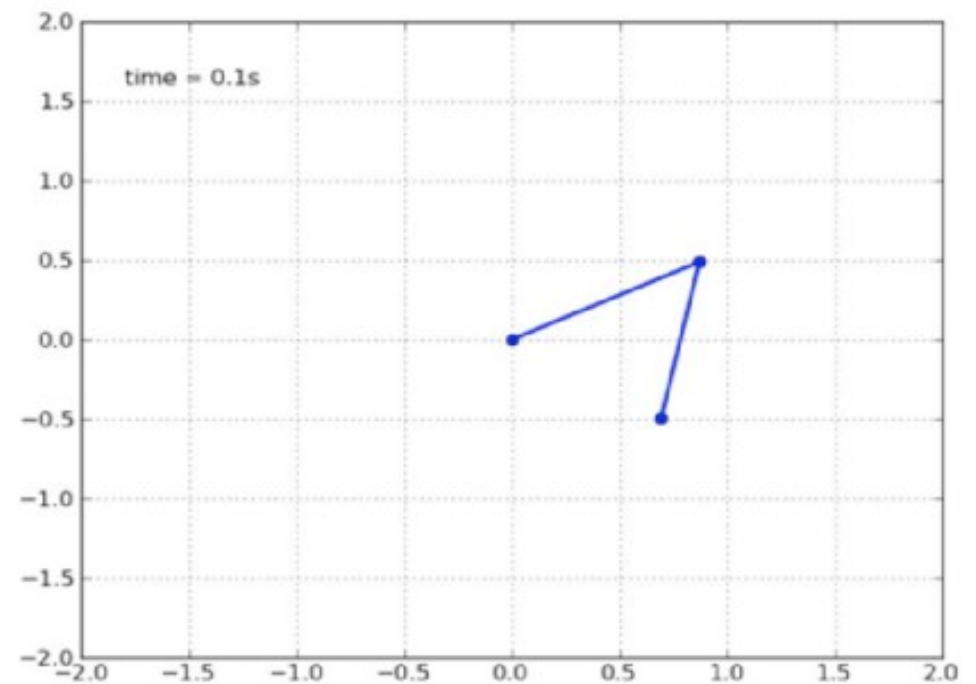
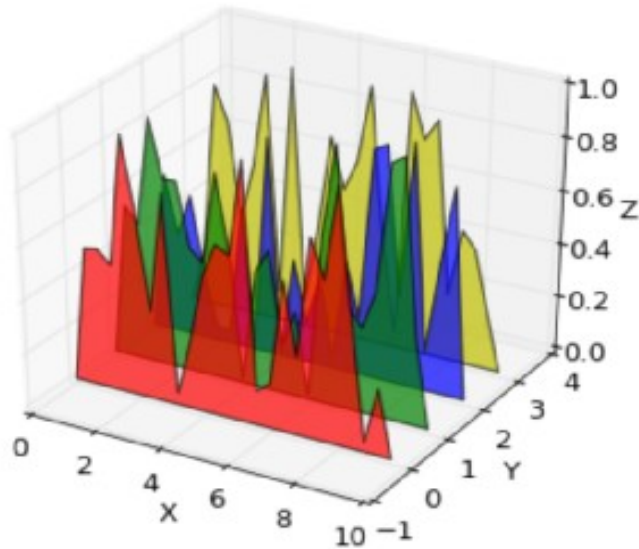
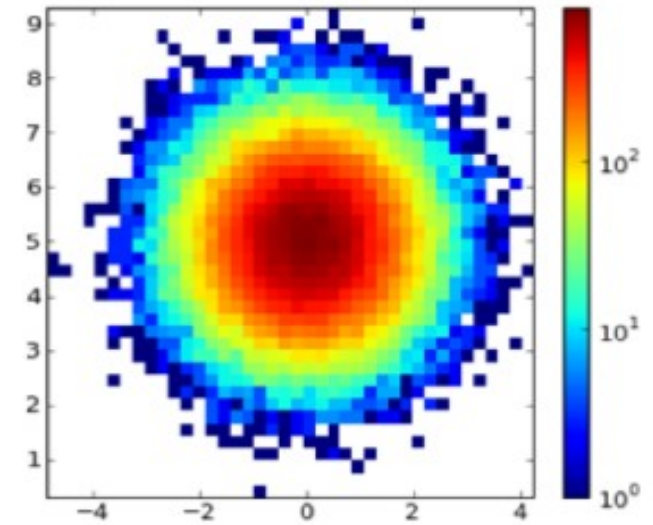
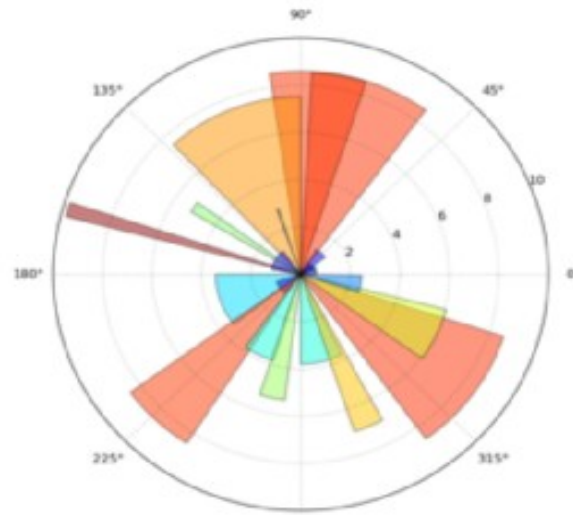
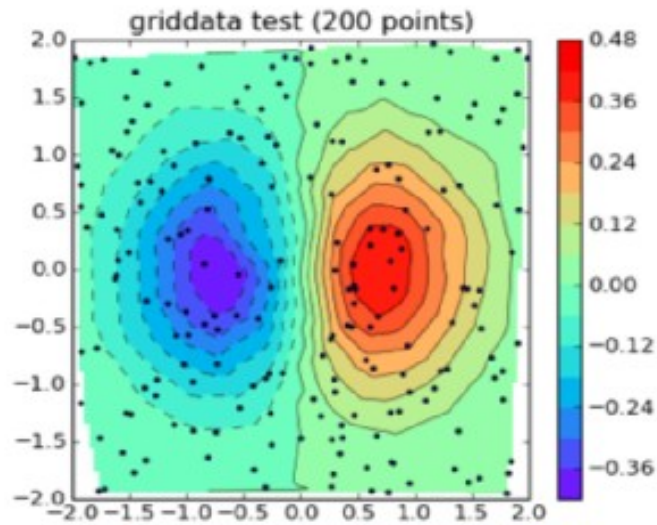
Forecasting



Large-scale physics  
simulations



# matplotlib



<http://matplotlib.org/>

# Games

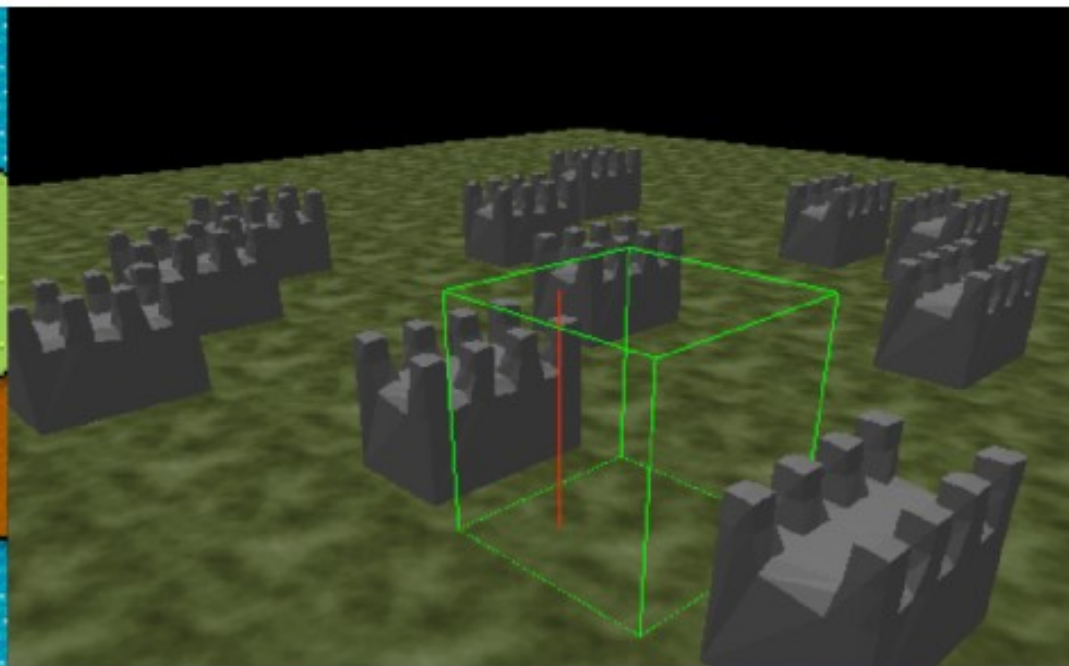
**EVE**<sup>®</sup>  
ONLINE







<http://pygame.org>





# Graphics



INDUSTRIAL LIGHT & MAGIC



"Python plays a key role in our production pipeline. Without it a project the size of Star Wars: Episode II would have been very difficult to pull off. From crowd rendering to batch processing to compositing, Python binds all things together." - Tommy Burnette, Senior Technical Director



# Finance



Python powers the New York Stock Exchange's web based transaction system.



# Business



“CORE” CRM and ERP  
platform

**NOKIA**

High-level programming  
environment for Symbian



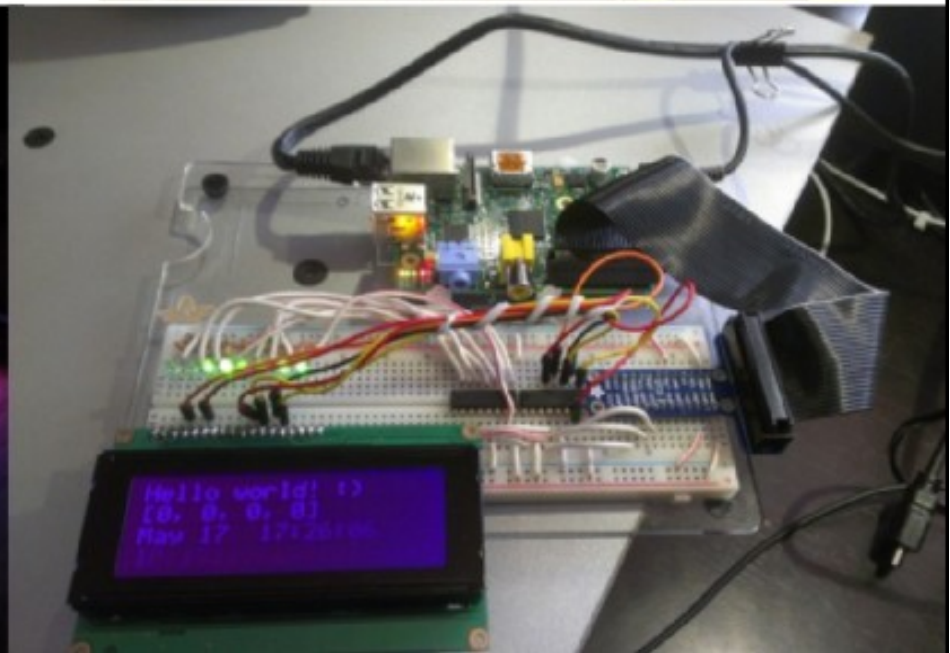
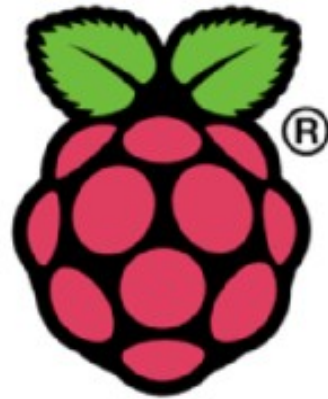
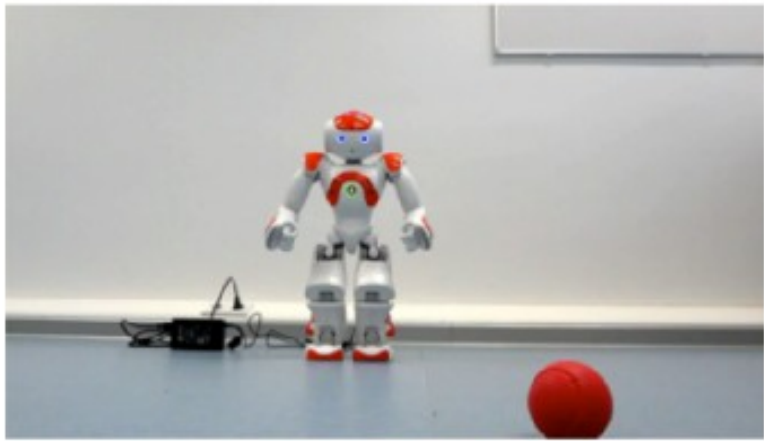
Desktop client



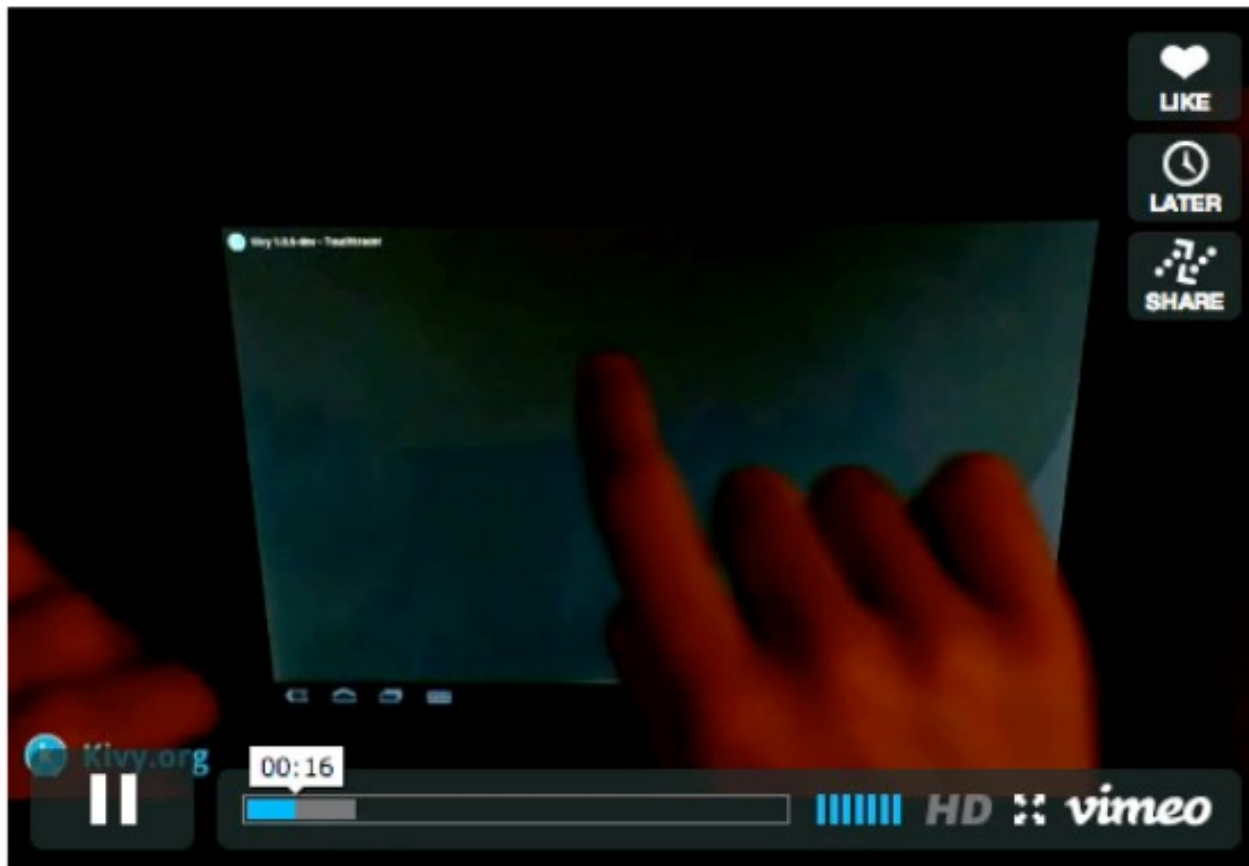
Installer and configuration  
utilities



# Hardware



# Even mobile



A multi-touch Android app written in Kivy.



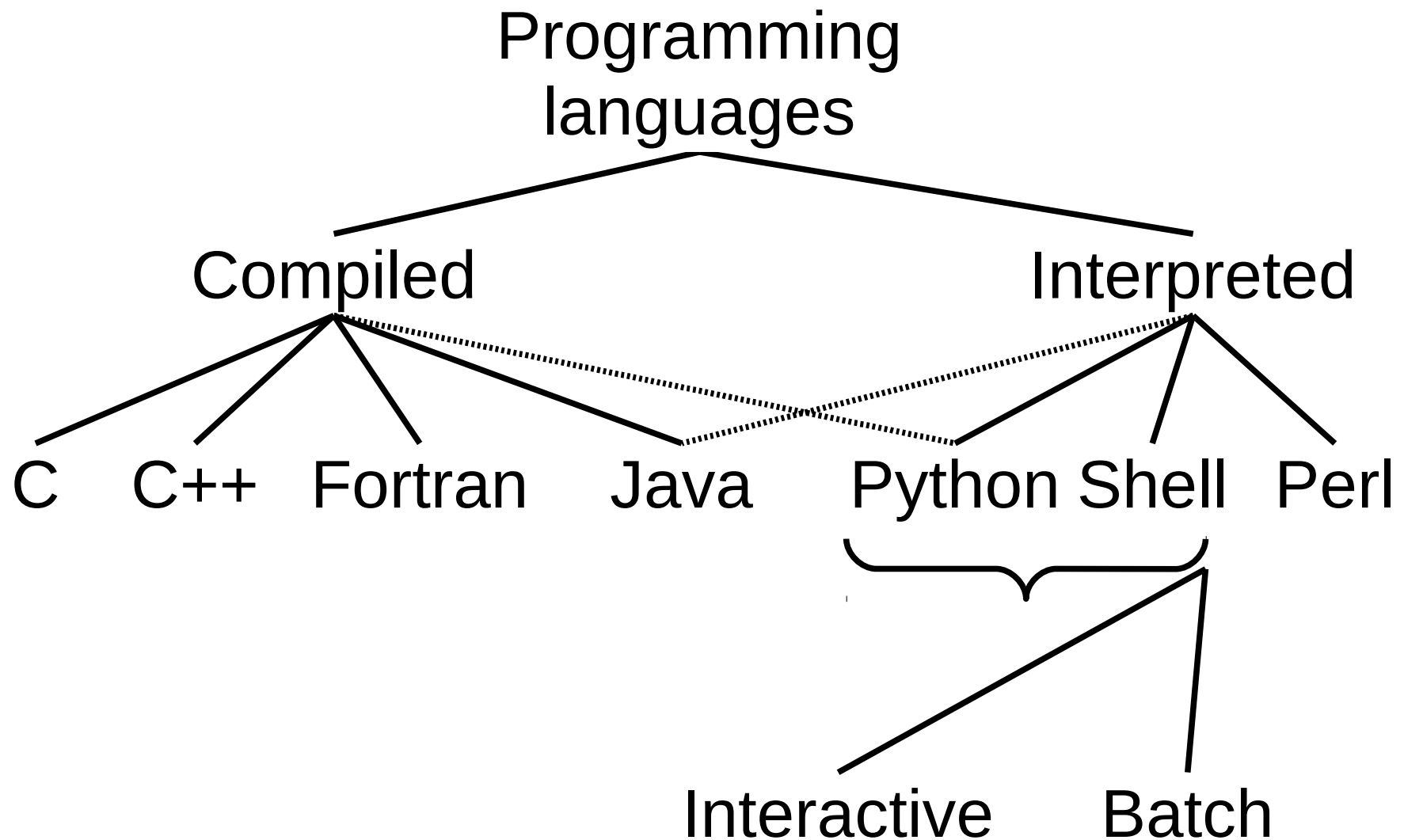


# Python is fast enough...

- CPython: the default / reference
  - implementation for the Python bytecode interpreter
- Written in C!

...and the ecosystem makes  
it even faster

- PyPy: a Python implementation with a Just-In-Time compiler
- C extensions
- Highly-optimized domain libraries





**Python 2 or Python 3 ?**



**\$ python**

Python 2.7.5 (default, Nov 12 2013, 16:45:58)  
[GCC 4.8.2 20131017 (Red Hat 4.8.2-1)] on linux2  
Type "help", "copyright", "credits" or "license"...

**>>> print 'Hello, world!'**

Hello, world!

**>>> 5**

3

**>>>**

To quit the Python interpreter:  
Press *control+d*

**\$**

Unix prompt

# Batch use

```
#!/usr/bin/python  
print 'Hello, world!'
```

hello.py

**\$ python hello.py**

Hello, world!

# Data types

- Numbers/Integers
- String
- List
- Dictionary
- Tuple

$\mathbb{Z}$

Integers

$\{ \dots -2, -1, 0, 1, 2, 3, \dots \}$



```
>>> 7+3
```


```
10
```

```
>>> 7*3
```

```
21
```

```
>>> 7/3
```

```
2
```



```
>>> 7%3
```

```
1
```

```
>>> 7-3
```


```
4
```

```
>>> 7**3
```

```
343
```

```
>>> -7/3
```

```
-3
```



```
>>> -7%3
```

```
2
```

$7^3$ : use “\*\*” for  
exponentiation

integer division  
rounds down

remainder (mod)  
returns 0 or positive  
integer

>>> **2\*2**

4

>>> **4\*4**

16

>>> **16\*16**

256


>>> **256\*256**

65536

>>> **65536\*65536**

4294967296L

“large” integer



```
>>> 4294967296*4294967296
```

```
18446744073709551616L
```

```
>>> 18446744073709551616 *
```

```
18446744073709551616
```

```
340282366920938463463374607431768211456L
```

```
>>> 2**521 - 1
```

```
6864797660130609714981900799081393217269
```

```
4353001433054093944634591855431833976560
```

```
5212255964066145455497729631139148085803
```

```
7121987999716643812574028291115057151L
```

No inherent limit to Python's integer arithmetic:  
can keep going until we run out of memory

2

4

16

C: int  
Fortran: INTEGER\*4

256

65536

C: long  
Fortran: INTEGER\*8

4294967296

Beyond the reach  
of C or Fortran

18446744073709551616





Floating  
point  
numbers

>>> **1.0**

1.0

Floating point number

>>> **0.5**

0.5

$\frac{1}{2}$  is OK

>>> **0.25**

0.25

$\frac{1}{4}$  is OK

Powers  
of two

>>> **0.1**

0.10000000000000000001

$\frac{1}{10}$  is *not*

Usual issues with representation in base 2

```
>>> 2.0*2.0
```

```
4.0
```

```
>>> 4.0*4.0
```

```
16.0
```

```
...
```

```
>>> 65536.0*65536.0
```

```
4294967296.0
```

```
>>> 4294967296.0*4294967296.0
```

```
1.8446744073709552e+19
```

17 significant figures

>>> 4294967296.0\*4294967296.0

1.8446744073709552e+19

>>> 1.8446744073709552e+19\*1.8446744073709552e+19

3.4028236692093846e+38

>>> 3.4028236692093846e+38\*3.4028236692093846e+38

1.157920892373162e+77

>>> 1.157920892373162e+77\*1.157920892373162e+77

1.3407807929942597e+154

>>> 1.3407807929942597e+154\*1.3407807929942597e+154

inf

overflow

Limit at  $2^{1023}$



# Machine epsilon

```
>>> 1.0 + 1.0e-16
```

```
1.0
```

too small to make  
a difference

```
>>> 1.0 + 2.0e-16
```

```
1.00000000000000000002
```

large enough

```
>>> 1.0 + 1.1e-16
```

```
1.0
```

```
>>> 1.0 + 1.9e-16
```

```
1.00000000000000000002
```

**Spend the next few minutes using Python interactively to estimate machine epsilon – we'll write a Python program to do this for us a little later**

# Mathematical Functions

Function	Returns ( description )
<a href="#"><u>abs(x)</u></a>	The absolute value of x: the (positive) distance between x and zero.
<a href="#"><u>ceil(x)</u></a>	The ceiling of x: the smallest integer not less than x
<a href="#"><u>cmp(x, y)</u></a>	-1 if $x < y$ , 0 if $x == y$ , or 1 if $x > y$
<a href="#"><u>exp(x)</u></a>	The exponential of x: $e^x$
<a href="#"><u>fabs(x)</u></a>	The absolute value of x.
<a href="#"><u>floor(x)</u></a>	The floor of x: the largest integer not greater than x
<a href="#"><u>log(x)</u></a>	The natural logarithm of x, for $x > 0$
<a href="#"><u>log10(x)</u></a>	The base-10 logarithm of x for $x > 0$ .
<a href="#"><u>max(x1, x2,...)</u></a>	The largest of its arguments: the value closest to positive infinity
<a href="#"><u>min(x1, x2,...)</u></a>	The smallest of its arguments: the value closest to negative infinity
<a href="#"><u>modf(x)</u></a>	The fractional and integer parts of x in a two-item tuple. Both parts have the same sign as x. The integer part is returned as a float.
<a href="#"><u>pow(x, y)</u></a>	The value of $x^{**}y$ .
<a href="#"><u>round(x [,n])</u></a>	x rounded to n digits from the decimal point. Python rounds away from zero as a tie-breaker: round(0.5) is 1.0 and round(-0.5) is -1.0.
<a href="#"><u>sqrt(x)</u></a>	The square root of x for $x > 0$

# Multiple assignments in a single line

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



# format

**a=3**

**b='python'**

**print "{} version is {}".format(b,a)**

# Strings

'Hello, world!'

'''Hello,  
world!'''

""Hello, world!""

""""Hello,  
world!""""

# Single quotes

'Hello, world!'

Single quotes around  
the string

# Double quotes

"Hello, world!"

Double quotes around  
the string

Exactly equivalent

**"He said "Python" is awesome"**

```
>>> print 'He said "Python!"is awesome'  
He said "Python!"is awesome
```

**"He said 'Python!' Is awesome"**

```
>>> print "He said 'Python!' is awesome"  
He said 'Python!' is awesome
```



# String concatenation

Two separate strings

>>> **'He said'** **'something to her.'**

'He saidsomething to her.'

Optional space(s)

>>> **'He said'"something to her.'**

'He saidsomething to her.'

>>> **'He said' + 'something to her.'**

'He saidsomething to her.'

Can also use + operator

# Special characters

`\n` → ↵

`\t` → →|

`\a` → 🎵

`\\` → \

`\'` → '

`\''` → ''

```
>>> print 'Hello,\nworld!'
```

```
Hello,  
world!
```

“\n” converted  
to “new line”

# Long strings

Triple double quotes

""" Long pieces of  
text are easier to  
handle if literal new  
lines can be  
embedded in them. """

# Long strings

Triple single quotes

'''Long pieces of text are easier to handle if literal new lines can be embedded in them.'''

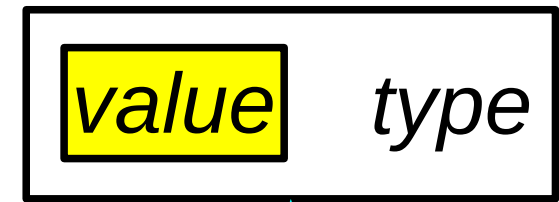


# String Methods Available

**title()  
upper()  
lower()  
swapcase()  
isalpha()  
isdigit()  
islower()  
isupper()  
istitle()  
split()**

**strip()  
lstrip()  
rstrip()  
find()  
startswith()  
endswith()  
replace()**

# How Python stores values



Type is stored with the value



**repr()**

*value*

**print**

“prettified”  
output

**type()**

*type*

```
>>> print 1.2345678901234567  
1.23456789012
```

```
>>> type( 1.2345678901234567 )  
<type 'float'>
```

```
>>> repr( 1.2345678901234567 )  
'1.2345678901234567'
```

# Two other useful types

Complex

```
>>> (1.0 + 2.0j) * (1.5 + 2.5j)  
(-3.5+5.5j)
```

Boolean

```
>>> True and False  
False
```

```
>>> 123 == 234  
False
```

# Comparisons

```
>>> 1 == 2
```

```
False
```

```
>>> 1 < 2
```

```
True
```

```
>>> 1 >= 2
```

```
False
```

```
>>> 1 == 1.0
```

```
True
```

```
>>> 'abc' == 'ABC'
```

```
False
```

```
>>> 'abc' < 'ABC'
```

```
False
```

```
>>> 'abc' >= 'ABC'
```

```
True
```



... not equal to ...

```
>>> not 1 == 2
```

```
True
```

```
>>> not 'abc' == 'ABC'
```

```
True
```

```
>>> 1 != 2
```

```
True
```

```
>>> 'abc' != 'ABC'
```

```
True
```

# Conjunctions

```
>>> 1 == 2 and 3 == 3
```

False

```
>>> 1 == 2 or 3 == 3
```

True

Evaluate the following Python expressions in your head:

```
>>> 2 - 2 == 1 / 2
```

```
>>> True and False or True
```

```
>>> 1 + 1.0e-16 > 1
```

```
>>> 5 == 6 or 2 * 8 == 16
```

```
>>> 7 == 7 / 2 * 2
```

```
>>> 'AbC' > 'ABC'
```

Now try them interactively in Python and see if you were correct.

# Precedence

First

$x^{**}y$

$-6, +6$

$x/y, x*y, x\%y$

$x+y, x-y$

$x<y, x<=y, \dots$

$x \text{ in } y, x \text{ not in } y$

$\text{not } x$

$x \text{ and } y$

Last

$x \text{ or } y$

Arithmetic  
operations

Logical  
operations

# Flow control in Python: **if**

```
if x > 0.0 :
```

```
    print 'Positive'
```

compulsory

indentation

```
elif x < 0.0 :
```

```
    print 'Negative'  
    x = -1.0 * x
```

optional,  
repeatable

multiple lines  
indented

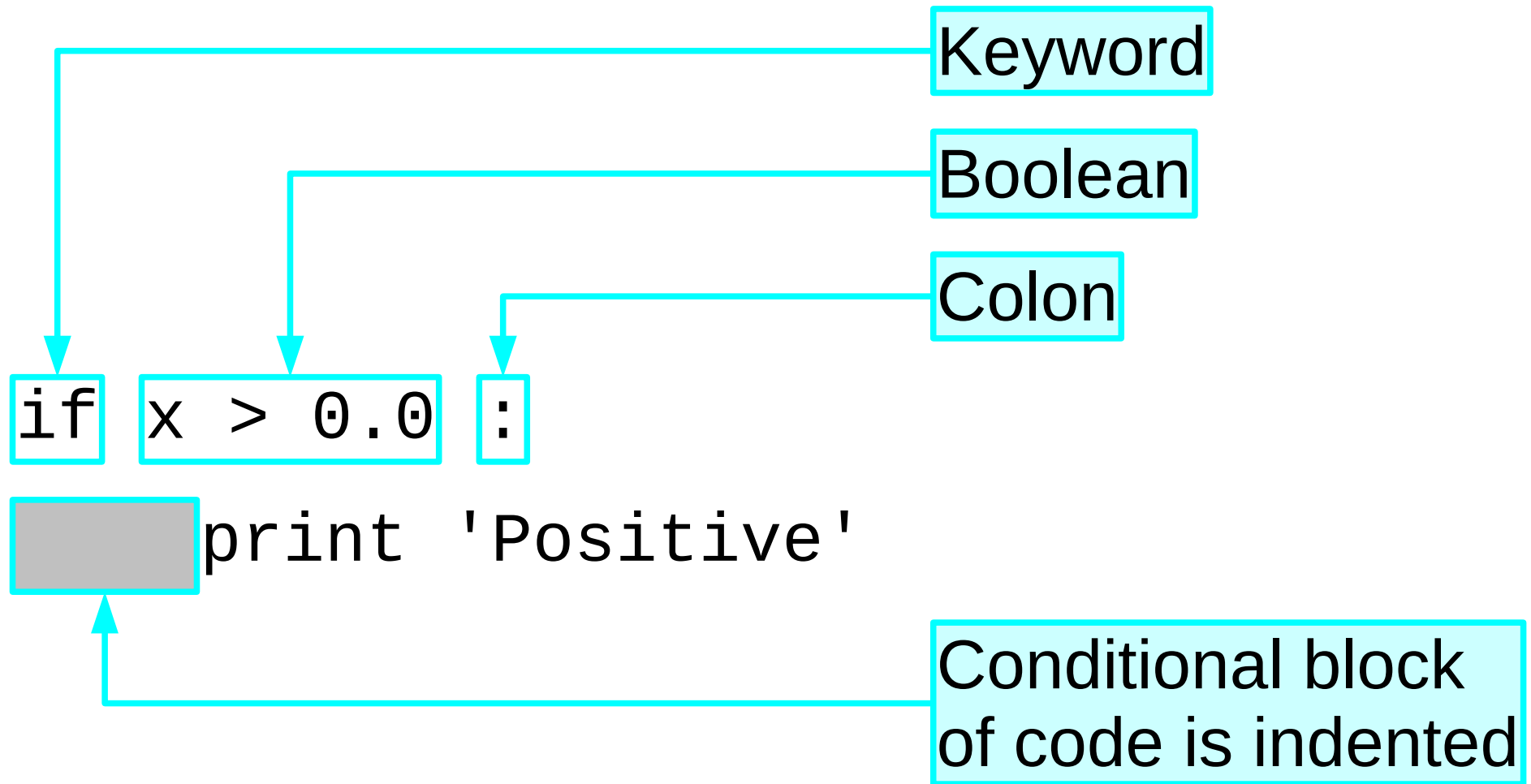
```
else :
```

```
    print 'Zero'
```

optional



# Flow control in Python: **if**



# Nested indentation

```
if x > 0.0 :  
    print 'Positive'  
else :  
    if x < 0.0 :  
        print 'Negative'  
        x = -1.0 * x  
    else :  
        print 'Zero'
```

# Flow control in Python: **while**

```
while x % 2 == 0 :
```

```
    print x, 'still even'
```

```
    x = x/2
```

compulsory

```
else :
```

```
    print x, 'is odd'
```

optional

# for loop

```
#!/usr/bin/python
for i in range(1, 5):
    print i
else:
    print 'The for loop is over'
```

# range()

range() is a built in class

```
range(stop)  
rang(start,stop)  
range(start, stop[, step])
```

# Remember Fibonacci series ?

1,1,2,3,5,8,13,21.....



**break** *statement*

**continue** *statement*

# Converting from one type to another

In and out of strings

```
>>> float('0.25')  
0.25
```



```
>>> str(0.25)  
'0.25'
```

```
>>> int('123')  
123
```



```
>>> str(123)  
'123'
```

# Converting from one type to another

Between numeric types

```
>>> int(12.3)  
12
```

loss of  
precision

```
>>> float(12)  
12.0
```

# Converting from one type to another

If you treat it like a list...

```
>>> list('abcd')
```

```
['a', 'b', 'c', 'd']
```

```
>>> list(data)
```

```
['line one\n', 'line two\n', 'line three\n', 'line four\n']
```

```
>>> list({'H':'hydrogen', 'He':'helium'})
```

```
['H', 'He']
```

# Time for a break...





January February March April May June July  
August September October November December

# Lists

H He Li Be B C N O F Ne Na Mg Al Si P S Cl Ar

Red Orange Yellow Blue Indigo Violet

```
>>> [ 2, 3, 5, 7, 11, 13, 17, 19]
```

```
[ 2, 3, 5, 7, 11, 13, 17, 19]
```

```
>>> type([ 2, 3, 5, 7, 11, 13, 17, 19])
```

```
<type 'list'>
```

```
>>> primes = [ 2, 3, 5, 7, 11, 13, 17, 19]
```

The diagram illustrates list indexing for the `primes` list. At the top, indices 0 through 7 are shown in light blue boxes. Cyan arrows point from each index to its corresponding prime value in the list `[2, 3, 5, 7, 11, 13, 17, 19]`. Below, the command `>>> primes[2]` is shown, with the result `5` displayed. A cyan arrow points from the index `2` in the command to the value `5` in the list, and another cyan arrow points from the value `5` in the list to the output `5`.

```
>>> primes = [ 2, 3, 5, 7, 11, 13, 17, 19]
```

```
>>> primes[2]
```

5

Indexing starts at 0

The diagram illustrates the relationship between indices and prime numbers. At the top, a row of eight light blue boxes contains the indices 0 through 7. Below these, a row of eight light blue boxes contains the corresponding prime numbers: 2, 3, 5, 7, 11, 13, 17, and 19. Cyan arrows point downwards from each index box to its respective prime number. Below the prime numbers, another row of eight light blue boxes contains the negative indices -8 through -1. Cyan arrows point upwards from each negative index box to the prime number it represents. To the left of the prime numbers, the text '>>> primes =' is shown. To the left of the negative indices, the text '>>> primes[-1]' is shown, with the value '19' printed below it.

```
>>> primes = [2, 3, 5, 7, 11, 13, 17, 19]
>>> primes[-1]
19
```

# Changing an item in a list

```
>>> data = [56.0, 49.5, 32.0]
```

```
>>> data[1]
```

49.5

“item number 1” (“2<sup>nd</sup> item”)

```
>>> data[1] = 42.25
```

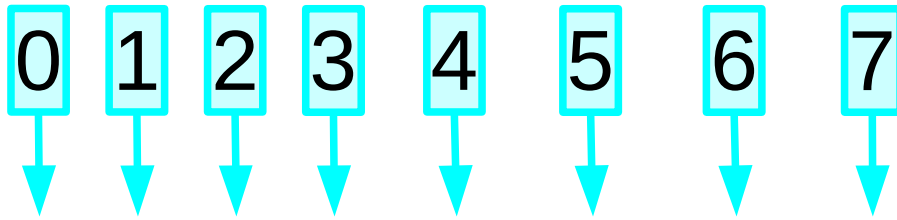
Assign new value to “item number 1” in list

```
>>> data
```

[56.0, 42.25, 32.0]

List is modified “in place”

0 1 2 3 4 5 6 7




```
>>> primes = [ 2, 3, 5, 7, 11, 13, 17, 19]
```


```
>>> primes[8]
```

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: list index out of range
```

Where the  
error was.



The error  
message.

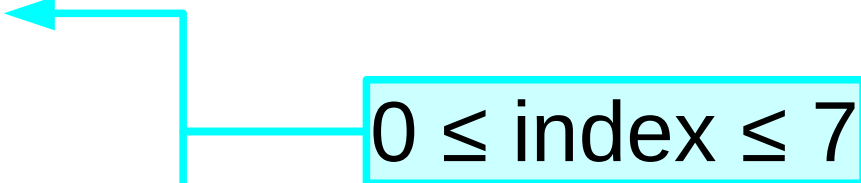


# Counting from zero and the `len()` function

```
>>> primes = [2, 3, 5, 7, 11, 13, 17, 19]
```

```
>>> primes[0]
```

2




A cyan line starts from the right side of the `primes[0]` expression, goes down, then right, then up, ending with an arrow pointing to the `primes[7]` expression. A cyan box containing the text `0 ≤ index ≤ 7` is connected to this line.

```
>>> primes[7]
```

19

```
>>> len(primes)
```

8



A cyan line starts from the right side of the `len(primes)` expression and goes left, ending with an arrow pointing to the output `8`. A cyan box containing the text `length 8` is connected to this line.



# Empty lists

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

```
>>> len(empty)  
0
```

```
>>> len([])  
0
```



# Single item lists

*A list with one item is not the same as the item itself!*

```
>>> [1234] == 1234  
False
```

```
>>> type([1234])  
<type 'list'>
```

```
>>> type(1234)  
<type 'int'>
```

# Lists of anything

```
primes = [ 2, 3, 5, 7, 11, 13, 17, 19 ]
```

List of integers

```
names = [ 'Alice', 'Bob', 'Cathy', 'Dave' ]
```

List of strings

```
roots = [ 0.0, 1.57079632679, 3.14159265359 ]
```

List of floats

```
lists = [ [ 1, 2, 3 ], [5], [9, 1] ]
```

List of *lists*

# Mixed lists

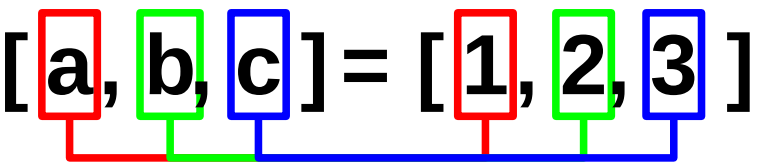
```
stuff = [ 2, 'Bob', 3.14159265359, 'Dave' ]
```



Legal, but not a good idea.  
See “tuples” later.

# Lists of variables

```
>>> [a, b, c] = [1, 2, 3]
```



The diagram illustrates the assignment of values to variables. The variables `a`, `b`, and `c` are enclosed in red, green, and blue boxes respectively. The values `1`, `2`, and `3` are also enclosed in red, green, and blue boxes respectively. Colored lines connect the boxes: a red line from `a` to `1`, a green line from `b` to `2`, and a blue line from `c` to `3`. The lines for `b` and `c` cross each other.

```
>>> a
```

1

```
>>> b
```

2

```
>>> c
```

3

# All or nothing

```
>>> [d, e, f] = [1, 2, 3, 4]
```

Traceback: where the error happened

```
graph TD; A[Traceback: where the error happened] --> B[Traceback (most recent call last):  
File "<stdin>", line 1, in <module>]; B --> C[ValueError: too many values to unpack]; C --> D[Error message]; D --> E[NameError: name 'd' is not defined];
```

Traceback (most recent call last):  
File "<stdin>", line 1, in <module>

ValueError: too many values to unpack

```
>>> d
```

Error message

Traceback (most recent call last):  
File "<stdin>", line 1, in <module>

NameError: name 'd' is not defined

# All or nothing

```
>>> [g, h, i, j] = [1, 2, 3]
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ValueError: need more than 3 values to unpack

```
>>> g
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'g' is not defined

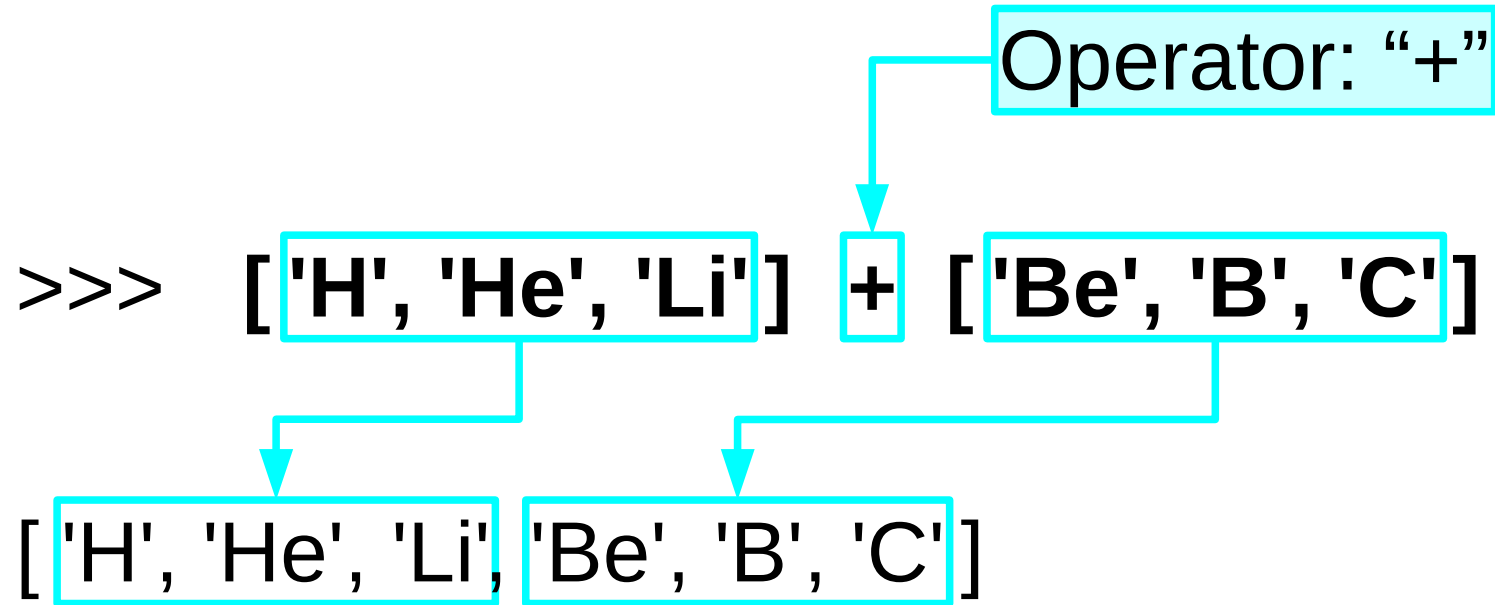
Error message



The diagram consists of two light blue boxes with black borders. The top box contains the text 'Error message'. Two arrows originate from this box. One arrow points upwards and to the left, ending at the error message 'ValueError: need more than 3 values to unpack' in the first code block. The other arrow points downwards and to the left, ending at the error message 'NameError: name 'g' is not defined' in the second code block.



# Concatenating lists



# Appending an item: **append()**

```
>>> symbols = [ 'H', 'He', 'Li', 'Be' ]
```

```
>>> symbols  
[ 'H', 'He', 'Li', 'Be' ]
```

appending is a “method”

the item to append

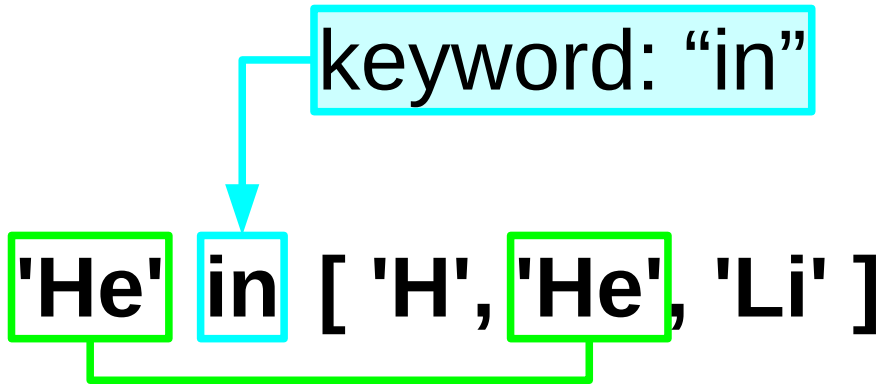
```
>>> symbols.append('B')
```

no value returned

```
>>> symbols  
[ 'H', 'He', 'Li', 'Be', 'B' ]
```

the list itself  
is changed

# Membership of lists

  
>>> 'He' in [ 'H', 'He', 'Li' ]

True

>>> 'He' in [ 'Be', 'B', 'C' ]

False

# Finding the index of an item

```
>>> symbols = [ 'H', 'He', 'Li', 'Be' ]
```

Finding the index is a method

the item to find

```
>>> symbols.index('H')
```

0

returns index of item

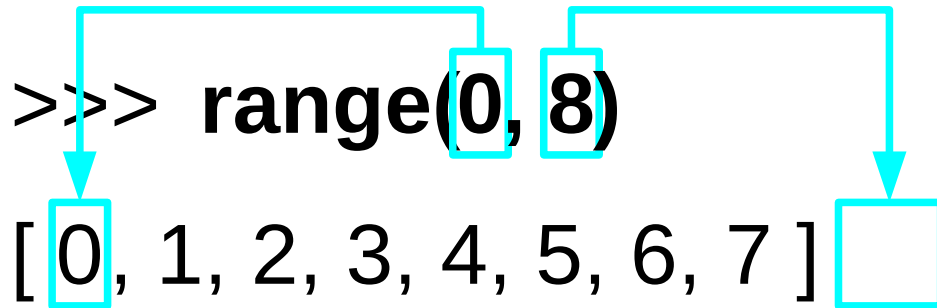
```
>>> metals = [ 'silver', 'gold', 'mercury', 'gold' ]
```

```
>>> metals.index('gold')
```

1

returns index of *first* matching item

# Functions that give lists: **range()**



>>> **range(0, 8)**

[0, 1, 2, 3, 4, 5, 6, 7 ]

The diagram illustrates the range function call and its output. The function call `range(0, 8)` is shown with the arguments `0` and `8` highlighted by cyan boxes. Arrows point from these boxes to the output list `[0, 1, 2, 3, 4, 5, 6, 7]`. The first element `0` and the last element `7` are also highlighted by cyan boxes. A cyan box is also present at the end of the list, indicating the range extends to the value 8.

First integer  
in list

One beyond  
last integer  
in list

# `range()`: Why miss the last number?

same argument

`>>> range(0, 8) + range(8, 12)`

`[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]`

`range(0, 12)`

# Functions that give lists: **split()**

original string

method built  
in to strings

>>> **'the cat sat on the mat'.split()**

**['the', 'cat', 'sat', 'on', 'the', 'mat']**

Split on white space

Spaces discarded

# **split()**: Only good for trivial splitting

```
>>> 'the cat sat on the mat'.split()
```

```
['the', 'cat', 'sat', 'on', 'the', 'mat']
```

Split on white space

Spaces discarded

Trivial operation

Regular expressions

Comma separated values

Use the specialist  
Python support  
for these.



list

method in  
every list

method takes  
an argument

>>> [ 'the', 'cat', 'sat', 'on', 'the', 'mat' ].count('the')

2

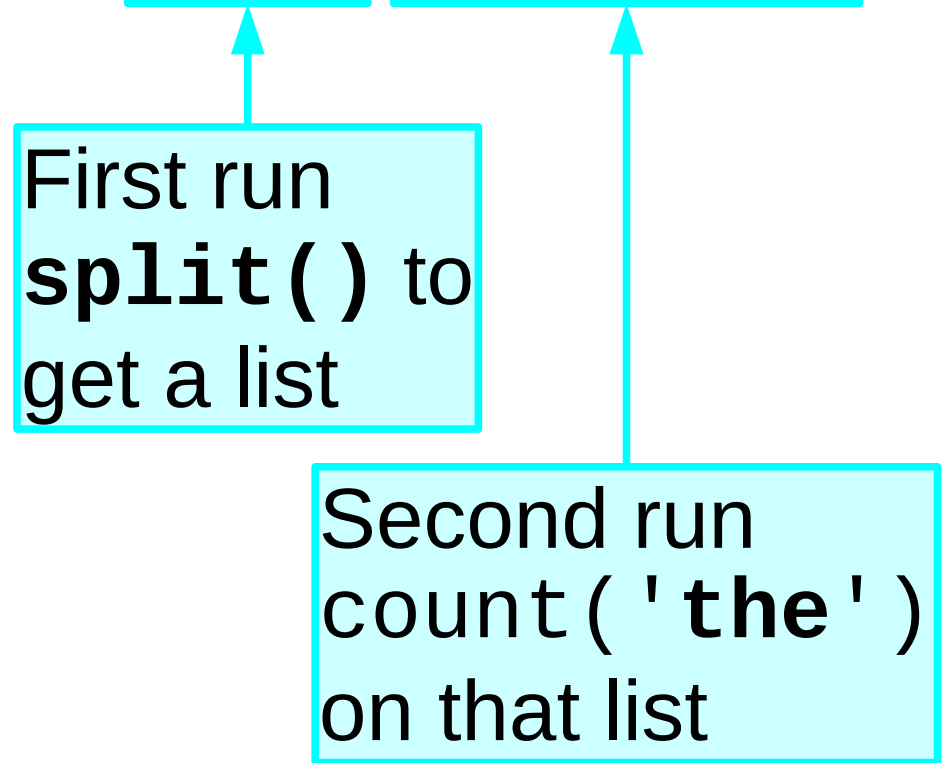
There are two  
'**the**' strings in  
the list.

# Combining methods

```
>>> 'the cat sat on the mat'.split().count('the')
```

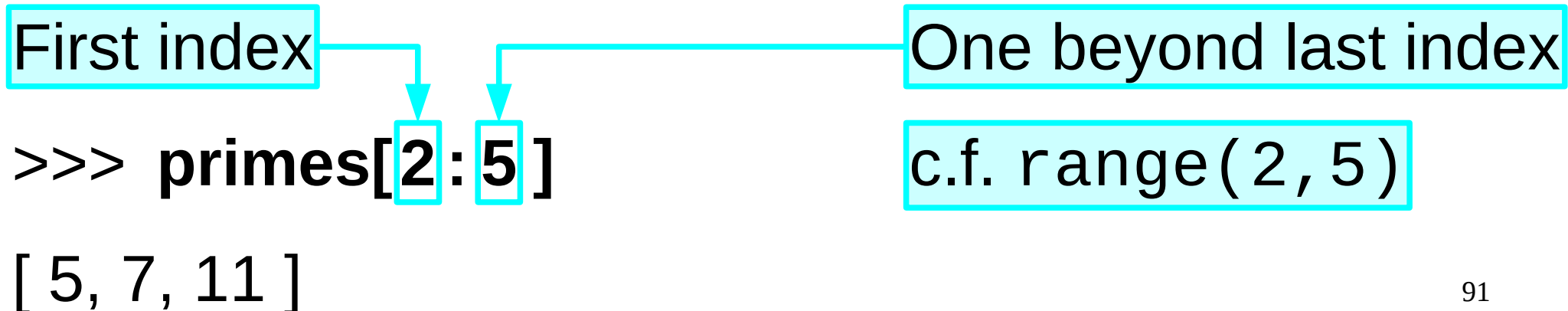
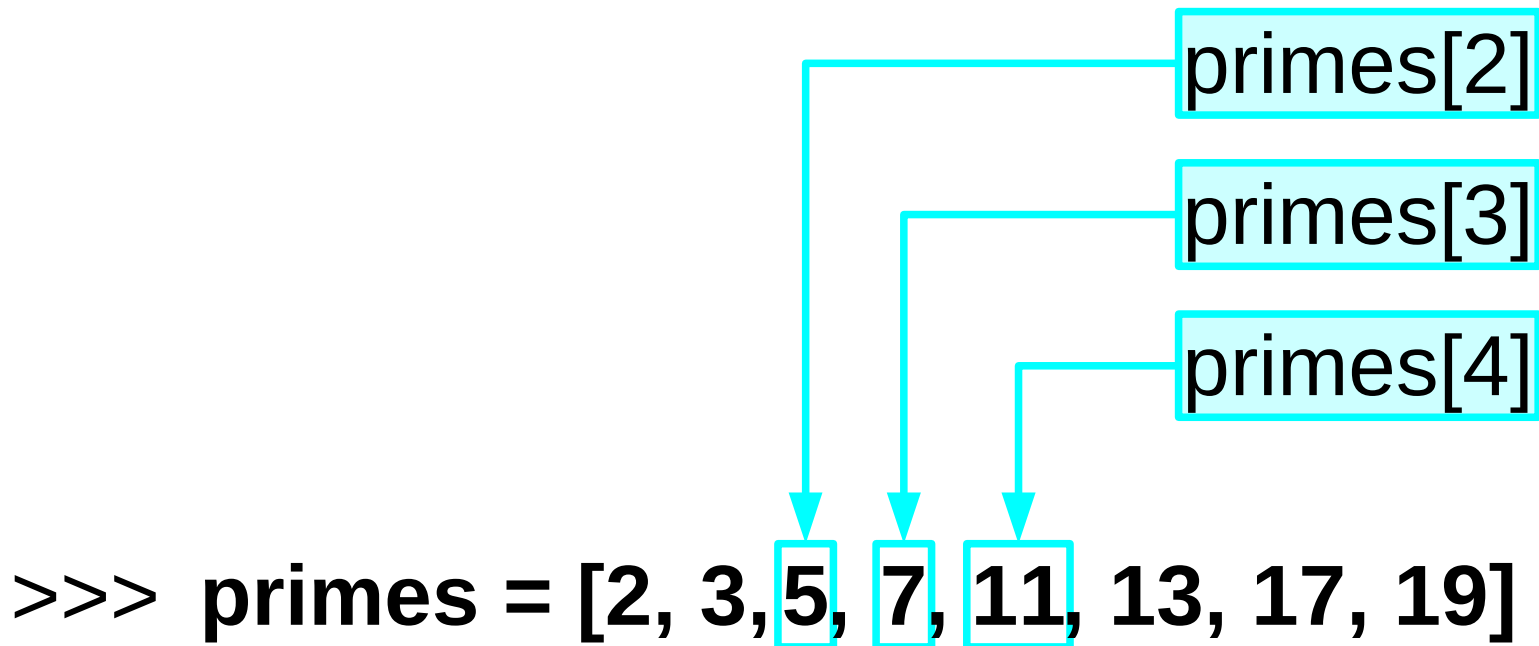
2

First run  
**split()** to  
get a list



Second run  
**count('the')**  
on that list

# Extracts from lists: “slices”



**>>> primes[2:5]**

**[ 5, 7, 11 ]**

Both limits given

**>>> primes[:5]**

**[ 2, 3, 5, 7, 11 ]**

Upper limit only

**>>> primes[2:]**

**[ 5, 7, 11, 13, 17, 19 ]**

Lower limit only

**>>> primes[:]**

**[ 2, 3, 5, 7, 11, 13, 17, 19 ]**

Neither limit given

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

```
# This is a list of some metallic  
# elements.
```

```
metals = [ 'silver', 'gold', ... ]
```

```
# Make a new list that is almost  
# identical to the metals list: the new  
# contains the same items, in the same  
# order, except that it does *NOT*  
# contain the item 'copper'.
```

**What goes here?**



```
# Print the new list.
```

metals.py

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

```
# This is a list of some data values.
```

```
data = [ 5.75, 8.25, ... ]
```

```
# Make two new lists from this list.
```

```
# The first new list should contain  
# the first half of data, in the same  
# order, whilst the second list should  
# contain the second half, so:
```

```
#     data = first_half + second_half
```

```
# If there are an odd number of items,  
# make the first new list the larger  
# list.
```

**What goes here?**

```
# Print the new lists.
```

data.py

```
#!/usr/bin/python

# This is a list of some metallic
# elements.
metals = [ 'silver', 'gold', ... ]

# Make a new list that is almost
# identical to the metals list: the new
# contains the same items, in the same
# order, except that it does *NOT*
# contain the item 'copper'.
new_metals = []
for metal in metals:
    if metal != 'copper':
        new_metals.append(metal)

# Print the new list.
print new_metals
```

```
#!/usr/bin/python
# This is a list of some data values.
data = [ 5.75, 8.25, ... ]
# Make two new lists from this list.
# The first new list should contain
# the first half of data, in the same
# order, whilst the second list should
# contain the second half, so:
#     data = first_half + second_half
# If there are an odd number of items,
# make the first new list the larger
# list.
if len(data) % 2 == 0:
    index = len(data) / 2
else:
    index = (len(data) + 1) / 2

first_half = data[:index]
second_half = data[index:]

# Print the new lists.
print first_half
print second_half
```



# Lists : More Methods

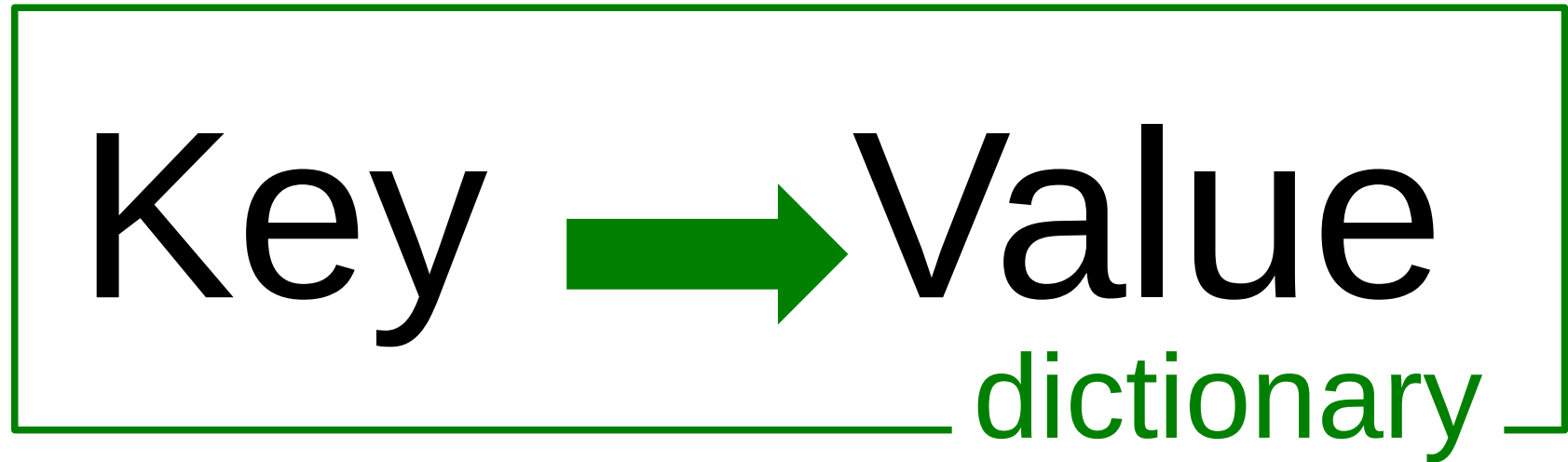
**insert()  
count()  
remove()  
reverse()  
sort()  
del a[-1]**

**pop()  
append()**

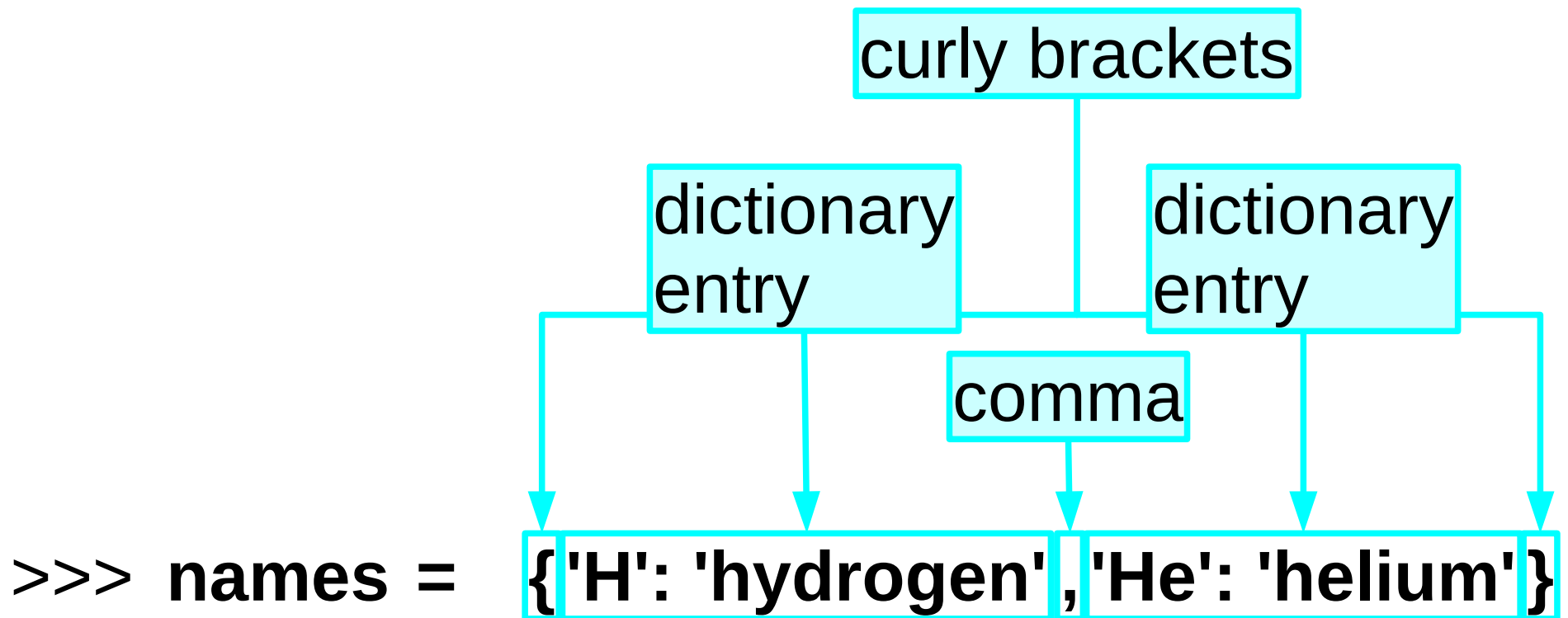
Using it as stack and queue



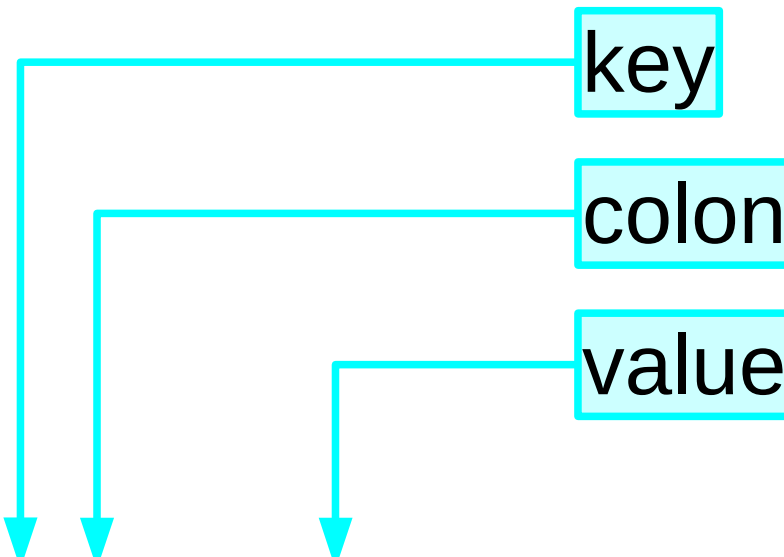
# Dictionaries



# Creating a dictionary — 1



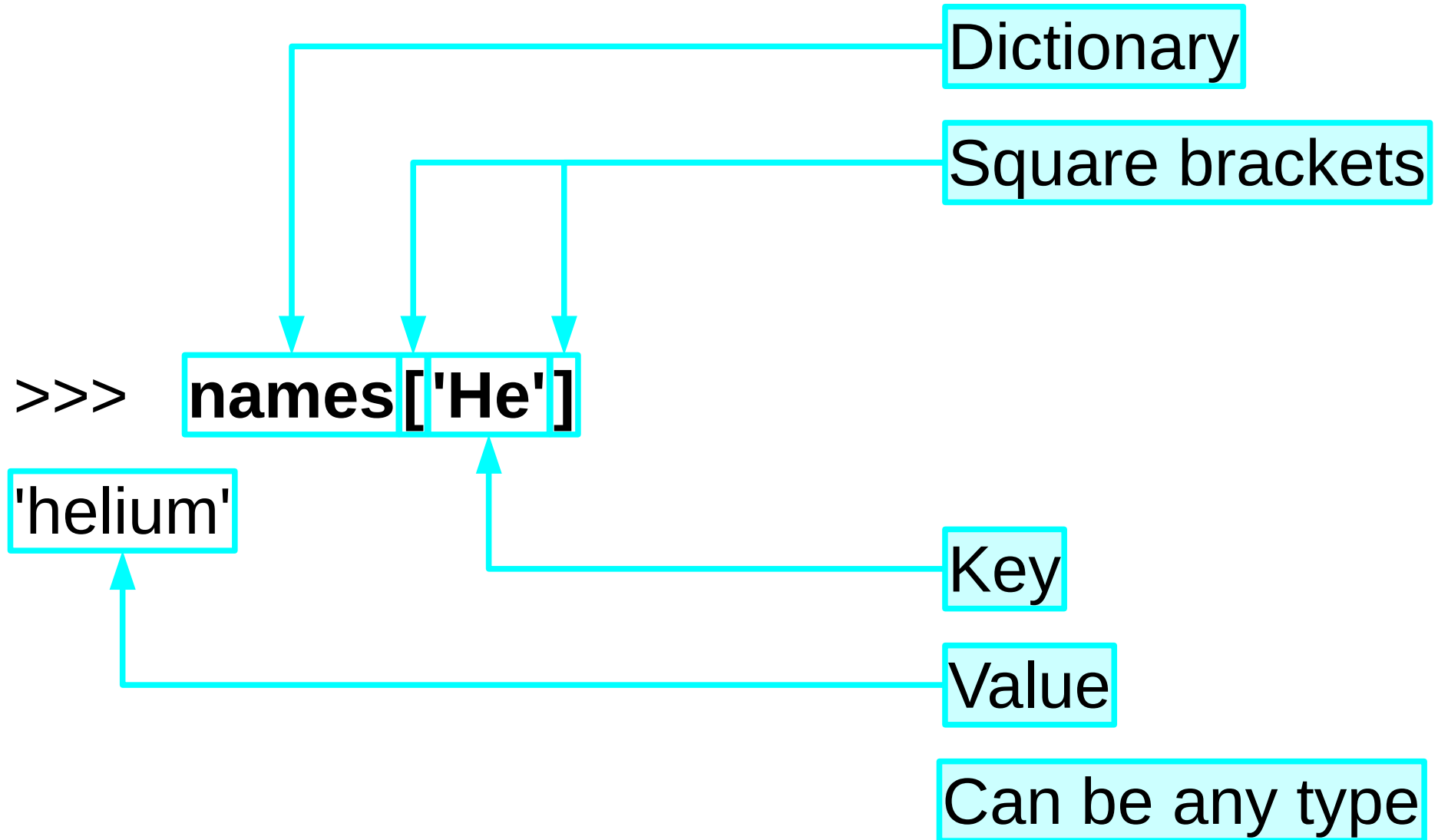
# Creating a dictionary — 2



```
>>> names = {'H': 'hydrogen', 'He': 'helium'}
```

The diagram illustrates the components of a dictionary key-value pair. Three labels in cyan boxes—'key', 'colon', and 'value'—have arrows pointing to the corresponding parts of the first pair in the code snippet: 'H' (key), ':' (colon), and 'hydrogen' (value). The entire code snippet is also enclosed in a cyan box.

# Accessing a dictionary



# Creating a dictionary — 3

**>>> names = {}** ← Start with an empty dictionary

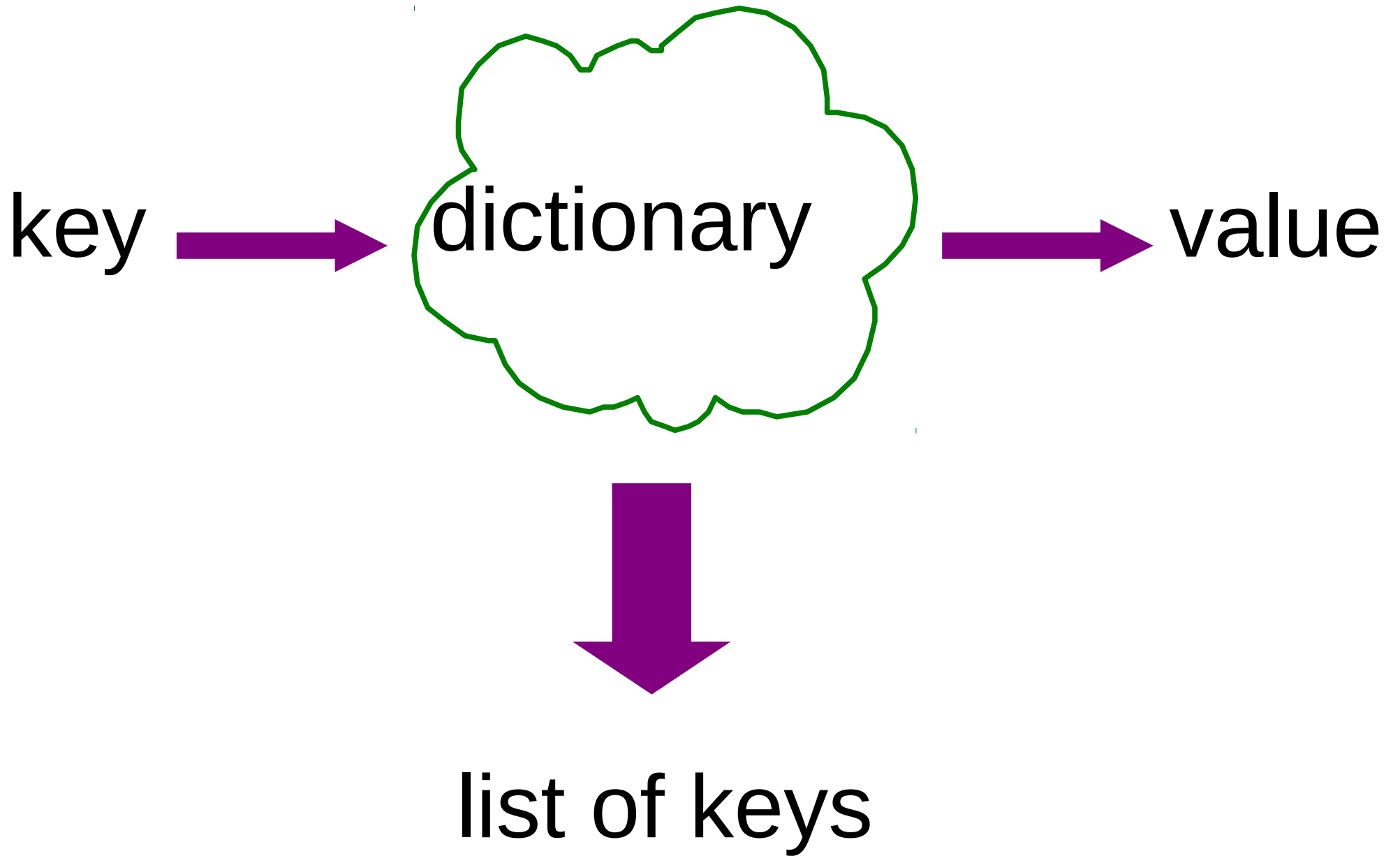
**>>> names[ 'H' ] = 'hydrogen'**

**>>> names[ 'He' ] = 'helium'**

**>>> names[ 'Li' ] = 'lithium'**

**>>> names[ 'Be' ] = 'beryllium'**

Add  
entries



# Treat a dictionary like a list...

Python expects a list here

```
for symbol in names:  
    print symbol, names[ symbol ]  
del symbol
```

Dictionary key

...and it behaves like a list of keys



# Example

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

```
names = {  
    'H': 'hydrogen',  
    'He': 'helium',  
    ...  
    'U': 'uranium',  
}
```

```
for symbol in names:  
    print names[symbol]  
del symbol
```

chemicals.py

```
$ python chemicals.py
```

ruthenium

rhenium


...

astatine

indium


No relation between  
order in file and output!

# Missing keys

  
>>> names['Np']

Traceback (most recent call last):  
File "<stdin>", line 1, in <module>

KeyError: 'Np'

  
Type of  
error

  
Missing key

# Treat a dictionary like a list...

Python expects a list here



```
if symbol in names:  
    print symbol , names[ symbol ]
```

...and it behaves like a list of keys

# Missing keys

```
>>> names['Np']
```

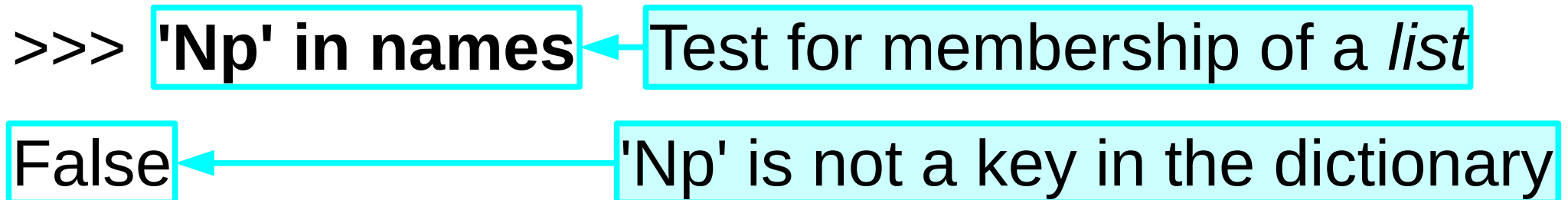
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
KeyError: 'Np'

```
>>> 'Np' in names
```

Test for membership of a *list*

False

'Np' is not a key in the dictionary



# Tuples

Singles  
Doubles  
Triples  
Quadruples  
Quintets

( 42 , 1.95 , 'Bob' )

( -1 , +1 )

( 'Intro. to Python', 25, 'TTR1' )

“not the same as lists”

# Tuples are not the same as lists

(minimum, maximum)

(age, name, height)

(age, height, name)

(age, height, name, weight)

Independent,  
grouped items

Related,  
sequential  
items

[ 2, 3, 5, 7 ]

[ 2, 3, 5, 7, 11 ]

[ 2, 3, 5, 7, 11, 13 ]

# Access to components

Same access syntax as for lists:

```
>>> ('Bob', 42, 1.95)[0]  
'Bob'
```

But tuples are *immutable*:

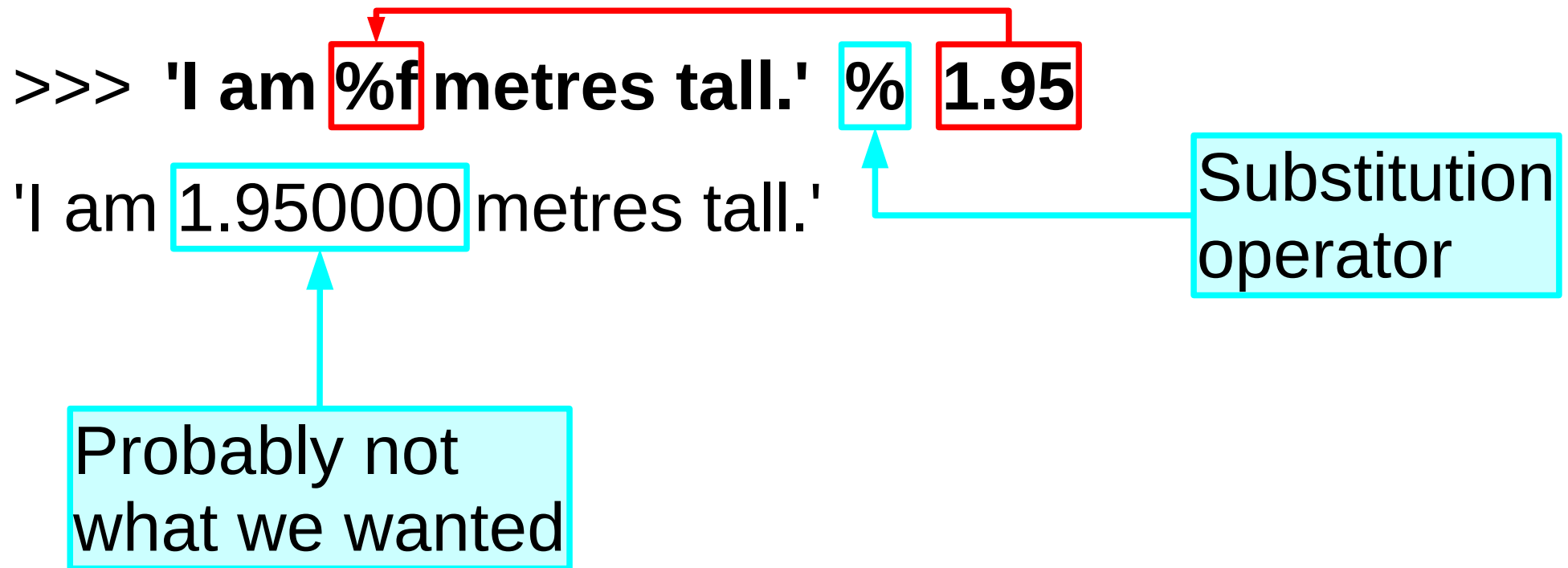
```
>>> ('Bob', 42, 1.95)[1] = 43
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'tuple' object does not support  
item assignment

# String substitution





# Substituting multiple values

```
>>> 'I am %f metres tall and my name is %s.'
```

```
% (1.95, 'Bob')
```

```
'I am 1.950000 metres tall and my name is Bob.'
```

# Formatted substitution

**>>> '%f' % 0.23**  
**'0.230000'**

standard float marker

six decimal places

**>>> '%.3f' % 0.23**  
**'0.230'**

modified float marker: “.3”

three decimal places

# More complex formatting possible

'23'	'23.4567'	'23.46'
' 23'	'23.456700'	'23.46 '
'0023'	'23.46'	' +23.46'
' +23'	' +23.4567'	' +23.46 '
' +023'	' +23.456700'	
'23 '	' +23.46'	' Bob '
' +23 '	'0023.46'	' Bob '
	' +023.46'	' Bob '

# Uses of tuples

1. Functions
2. Related data
3. String substitution



# And now for something completely...



# Defining functions

```
def functionname(params):  
    statement1  
    statement2
```

# Local and global variables



# Default argument values

```
def functionname(param1, param2 = value):  
    statement1  
    statement2
```

# Keyword arguments

```
def functionname(param1, param2 = value):  
    statement1  
    statement2
```

## **return** *statement*

```
def functionname(param1, param2 = value):  
    statement1  
    statement2  
    return "values are passsed"
```

# Classes

```
class classname:  
    body
```

self

# `__init__`

- runs as soon as an object of a class is instantiated
- useful to do any initialization

# Inheritance

- Reusing the code

```
class A:
```

```
class B(A):
```

```
class C(A)
```



# Inheritance

- Reusing the code

```
class A:
```

```
class B(A):
```

```
class C(A):
```

# Modules

```
Import modulename  
from modulename import *
```

profile getpass re bisect os gzip pickle time  
anydbm bz2  
calendar  
unittest  
atexit  
shelve  
datetime  
cgi csv  
asyncore  
optparse  
asynchat  
webbrowser  
mmap  
hmac  
BaseHTTPServer  
SimpleHTTPServer  
CGIHTTPServer  
math  
sched  
cmath  
heapq  
imageop  
email  
audioop  
Cookie  
logging  
sys  
unicodedata  
base64  
sets  
codecs  
stringprep  
tempfile  
hashlib  
mutex  
select  
string  
chunk  
code  
ConfigParser  
locale  
cmd  
linecache  
collections  
glob  
colorsys  
gettext

# System modules

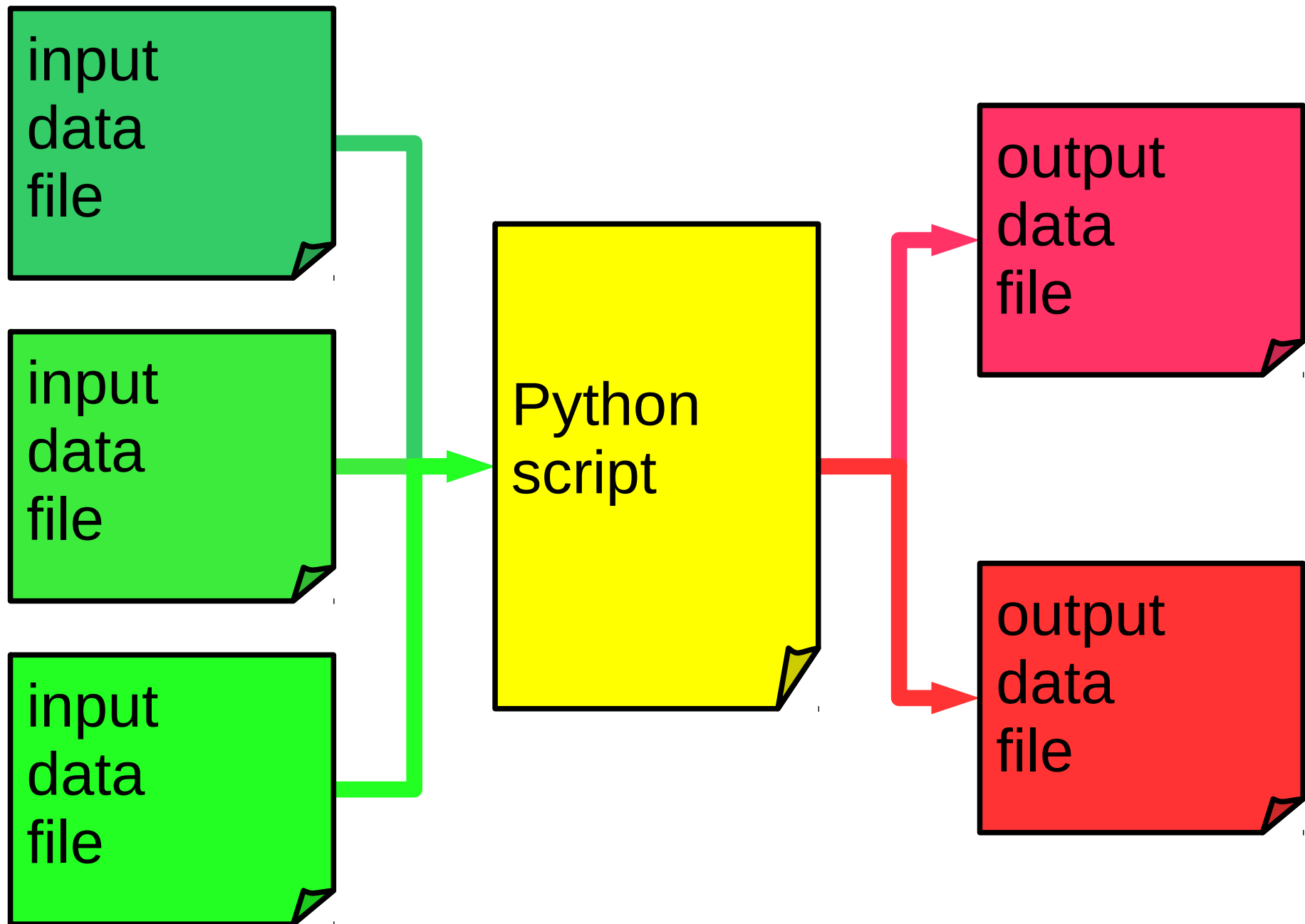
Let's break for an  
exercise...



# Accessing the system

1. Files
2. Standard input & output
3. The command line

# May want to access many files



```
line one\n  
line two\n  
line three\n  
line four\n
```

data.txt

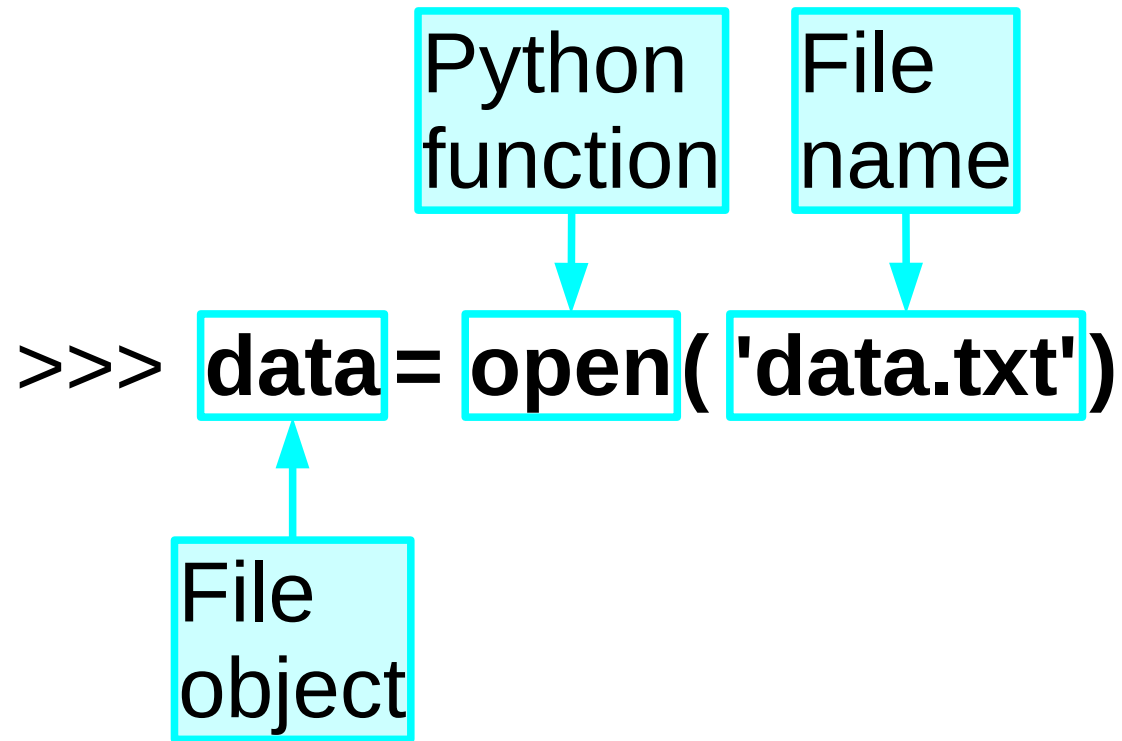
=

```
line one←line t  
wo←line three←  
line four←
```

data.txt

\n

←



All access to the file is via the file object



```
>>> data = open( 'data.txt' )
```

method to read a line

```
>>> data.readline()
```

'line one\n'

first line of file,  
complete with “\n”

```
>>> data.readline()
```

same command

'line two\n'

second line of file

```
>>> data = open( 'data.txt' )
```

```
>>> data.readline()
```

```
'line one\n'
```

```
>>> data.readline()
```

```
'line two\n'
```

```
>>> data.readlines()
```

```
['line three\n', 'line four\n']
```

remaining lines



```
>>> data = open( 'data.txt' )
```

```
>>> data.readline()
```

```
'line one\n'
```

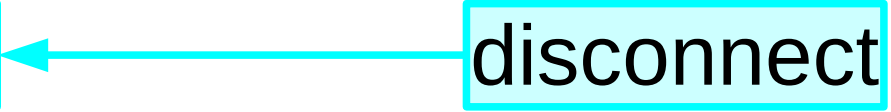
```
>>> data.readline()
```

```
'line two\n'
```

```
>>> data.readlines()
```

```
[ 'line three\n', 'line four\n' ]
```

```
>>> data.close()
```



disconnect

```
>>> del data
```

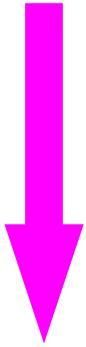


delete the variable

# Treating file objects like lists:

```
for line in data.readlines():  
    do stuff
```

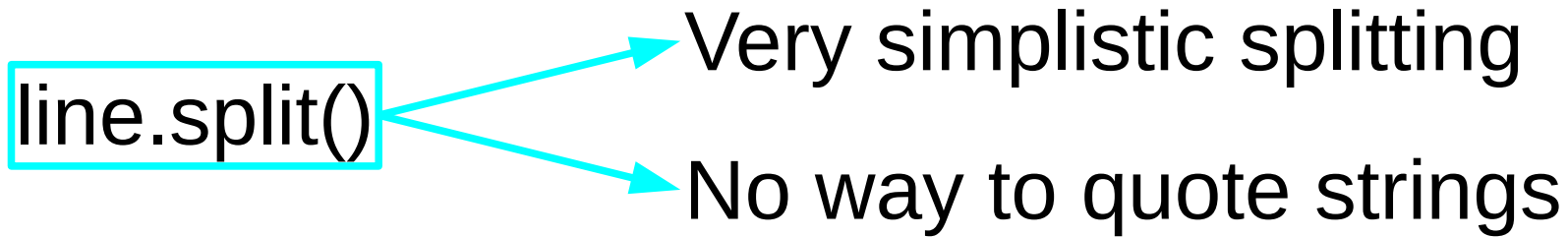
reads the lines  
all at once



```
for line in data:  
    do stuff
```

reads the lines  
as needed

# Very primitive input



Comma separated values:	<b>csv</b> module
Regular expressions:	<b>re</b> module

**“Python: Further Topics”** course

**“Python: Regular Expressions”** course

# Reading data gets you strings

```
1.0
2.0
3.0
4.0

four.dat
```

readlines() → ['1.0\n', '2.0\n', ...]

strings

*not* floats

```
>>> '1.0\n'.strip()
'1.0'
```


Method to clear  
trailing white space

Still need to convert string to other types

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

```
# This script reads in some  
# numbers from the file 'numbers.txt'.  
# It then prints out the smallest  
# number, the arithmetic mean of  
# the numbers, and the largest  
# number.
```

**What goes here?  
(Use the function  
you wrote in an  
earlier exercise.)**



# Output to files

```
>>> output = open('output.dat', 'w')
```

```
>>> output.write('alpha\n')
```

explicit “\n”

```
>>> output.write('bet')
```

write(): writes  
*lumps* of data

```
>>> output.write('a\n')
```

```
>>> output.writelines(['gamma\n', 'delta\n'])
```

```
>>> output.close()
```

Flushes to  
file system



# Command line

`import sys` ← `sys module`

`sys.argv` ← `list of arguments`

`sys.argv[0]` ← `name of script`

```
#!/usr/bin/python  
  
print sys.argv[0]  
  
print sys.argv
```

args.py

**\$ python args.py 0.25 10**

args.py

['args.py', '0.25', '10']

**NB:** list of *strings*

# Questions



