

(Python) Fundamentals



by
__Mani

overview

9:00 - 9:45 Programming basics

10:00 - 11:45 Python basics

12:00-12:45 Advanced Python

12:45-13:00 QA

... please ask questions!

Why Python?

- Why not Java, VBA, Perl, Ruby, C#, ...?

Simple Task

- Read a text file and print out non-comment lines

test.csv

```
#comment  
Bank,Account,Amount  
BAML,12345,$1000  
"Citi Bank",54321,S$500
```

Pseudocode:

```
open file test.csv  
    for each line in file  
        if line does not start with '#'  
            print line
```

Java

```
import java.io.BufferedReader;
import java.io.FileReader;

public class CsvFile {
    public static void main(String[] args) {
        String line = null;
        try{
            BufferedReader br = new BufferedReader(new FileReader("test.csv"));
            try {
                while ((line=br.readLine()) != null) {
                    if (! line.trim().startsWith("#")) {
                        System.out.println(line);
                    }
                }
            } finally {
                br.close();
            }
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

Perl & Ruby

- Perl

```
use strict;
use warnings;

open(my $data, 'test.csv') or
    die "Error reading test.csv
    $!\n";

while (my $line = <$data>) {
    if ($line !~ /^#/) {
        print $line;
    }
}
```

- Ruby

```
File.open("test.csv", "r") do
    |infile|
        while (line = infile.gets)
            if !
                line.start_with?('#')
                    puts "#{line}"
            end
        end
    end
end
```

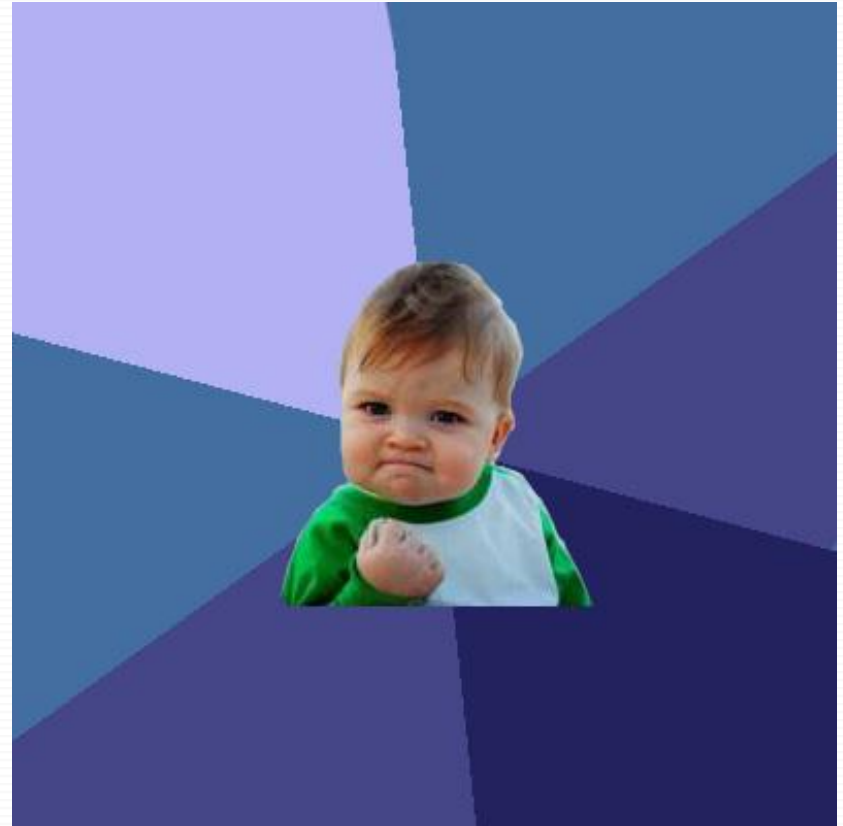
Python

- Python

```
with open('test.csv', 'r') as f:
    for l in f:
        if not l.startswith('#'):
            print l
```

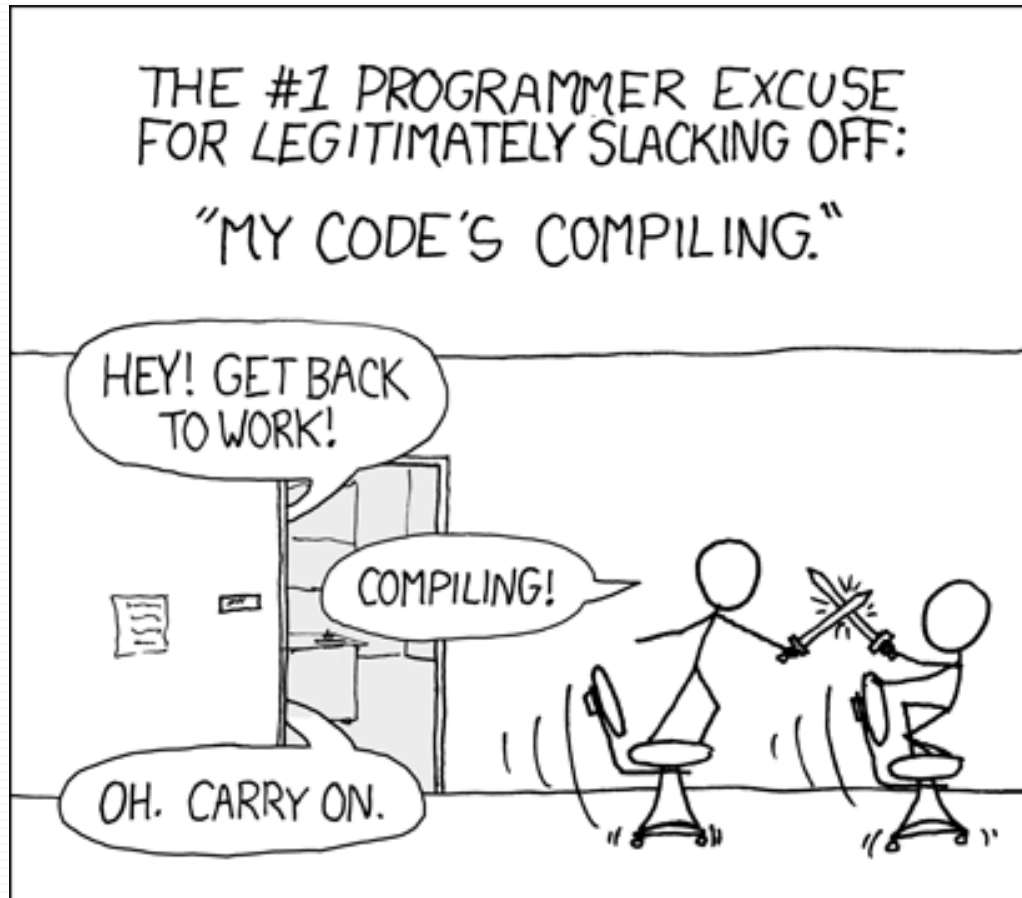
Pseudocode:

```
open file test.csv
for each line in file
    if line does not start with '#'
        print line
```



Python, is that all you got?

- No compilation needed, no build monitor



Python, is that all you got?

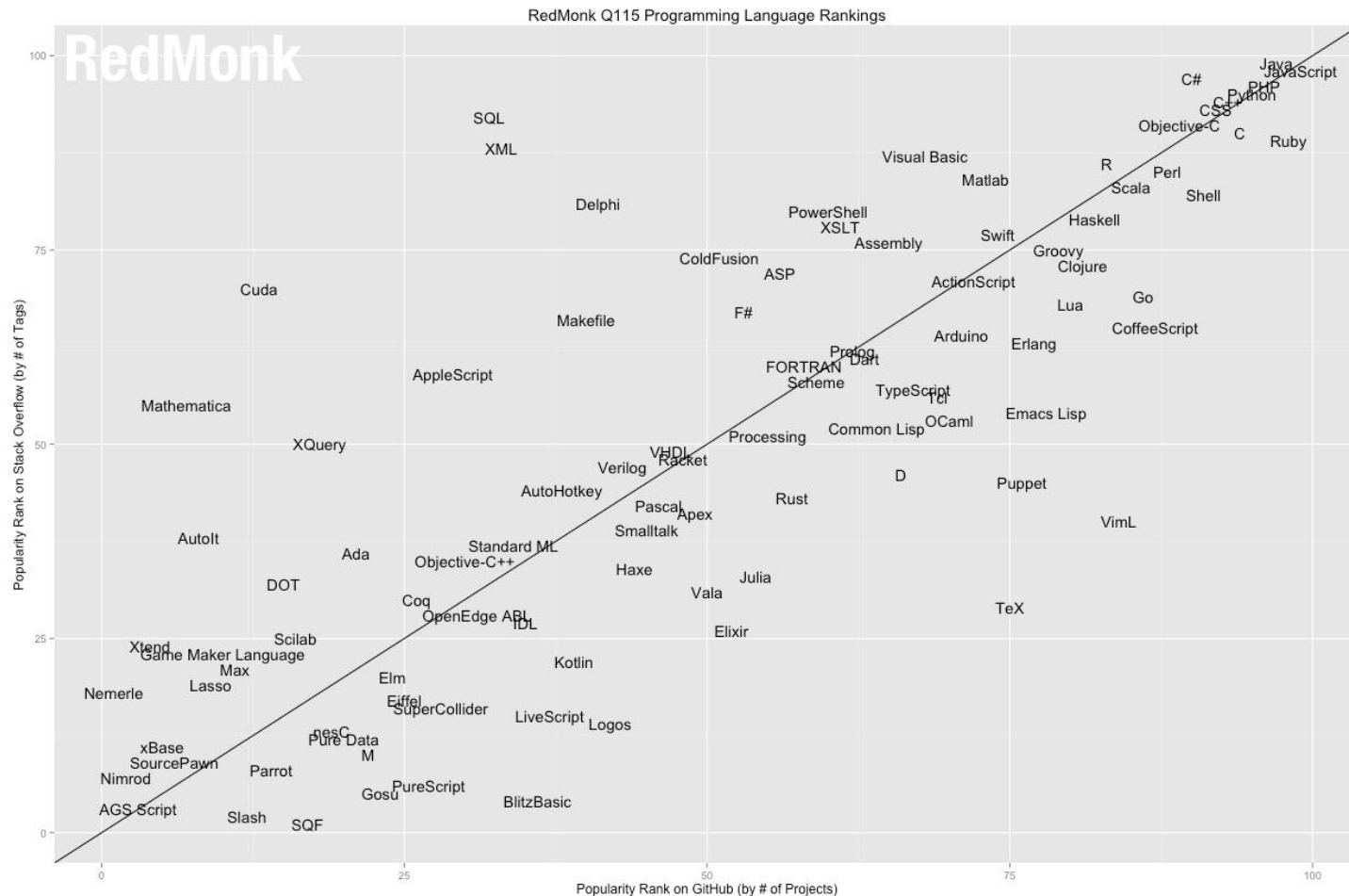
- Duck type

Don't check whether something **IS** a duck. Check whether it **QUACKS** like a duck and **WALKS** like a duck

-- Alex Martelli



Python Popularity



(Python) Fundamentals



simple data types

Values of different **type** allow different **operations**

12 + 3

=> 15

12 - 3

=> 9

"12" + "3"

=> "123"

"12" - "3"

=> ERROR!

"12"*3

=> "121212"

- **bool**: Boolean, e.g. **True**, **False**
 - **int**: Integer, e.g. **12**, **23345**
 - **float**: Floating point number, e.g. **3.1415**, **1.1e-5**
 - **string**: Character string, e.g. **"This is a string"**
 - ...
-

structured types

structured data types are composed of other simple or structured types

- ❑ **string**: 'abc'
- ❑ **list of integers**: [1,2,3]
- ❑ **list of strings**: ['abc', 'def']
- ❑ **list of lists of integers**: [[1,2,3],[4,5,6]]
- ❑ ...

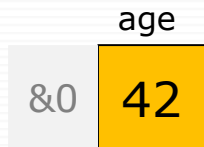
there are much more advanced data structures...

data structures are models of the objects you want to work with

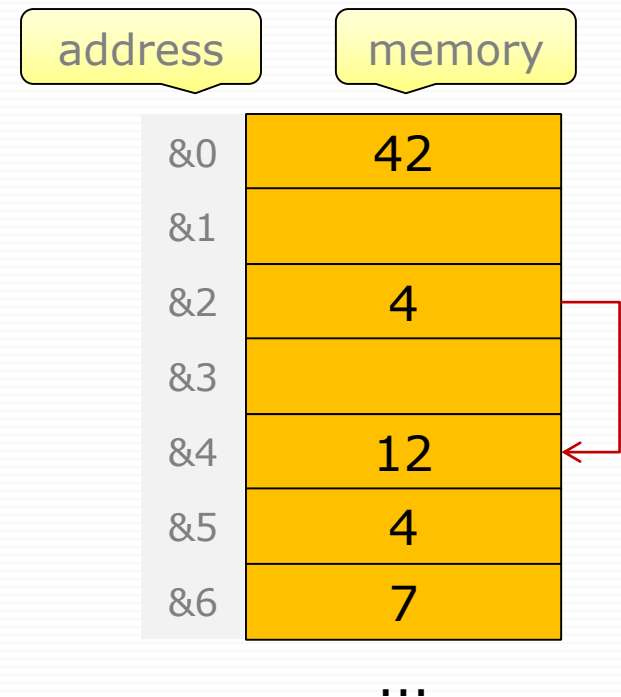
variables and references

- ❑ container for a value, name of a memory cell
- ❑ primitive values: numbers, booleans, ...
- ❑ reference values: address of a data structure, eg. a list, string, ...

```
age = 42
```



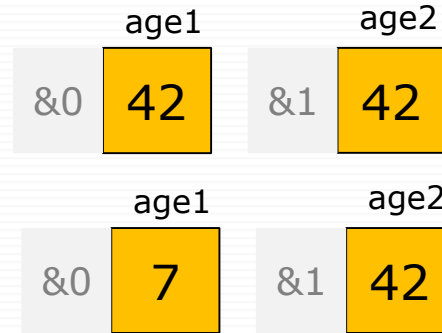
```
ages = [12,4,7]
```



variables and references 2

```
age1 = 42  
age2 = age1
```

```
age1 = 7    => age2 is 42
```

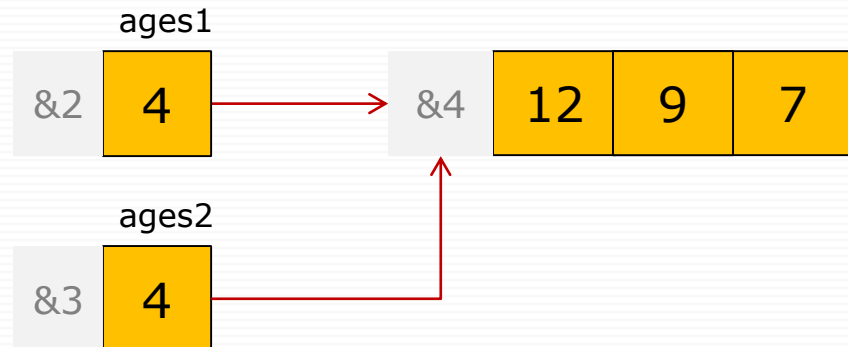


```
try:  
    id(age1)  
    id(age2)
```

```
ages1 = [12,4,7]  
ages2 = ages1
```

```
ages1[1] = 9
```

```
=> ages2[1] is 9 !
```



data structures 1

□ organizing data depending on task and usage pattern

□ List (Array)

- collection of (many) things
- constant access time
- linear search time
- changeable (=mutable)

```
colors = ['red', 'red', 'blue']
```

```
colors[1] => 'red'
```

```
'blue' in colors => True
```

```
colors[1] = "pink"
```

□ Tuple

- collection of (a few) things
- constant access time
- linear search time
- not changeable (=immutable)
=> can be used as hash key

```
address = (3, 'Queen Street')
```

```
address[0] => 3
```

```
3 in address => True
```

```
address[0] = 4
```


data structures 2

□ Set

- collection of unique things
- no random access
- constant search time
- no guaranteed order of values

```
even = set([2,4,6,8])
```

```
even[1]
```

```
6 in even                => True
```

```
for e in even: print e
```

□ Dictionary (Hash)

- maps keys to values
- constant access time via key
- constant search time for key
- linear search time for value
- no guaranteed order

```
tel = {'Stef':62606, 'Mel':62663}
```

```
tel['Mel']                => 62663
```

```
'Stef' in tel             => True
```

```
62663 in tel.values()     => True
```

```
for name in tel: print name
```

data structures - tips

when to use what?

□ List

- many similar items to store
e.g, numbers, protein ids, sequences, ...
- no need to **find** a specific item fast
- fast access to items at a specific **position** in the list

□ Tuple

- a few (<10), different items to store
e.g. addresses, protein id and its sequence, ...
- want to use it as dictionary key

□ Set

- many, unique items to store
e.g. unique protein ids
- need to know quickly if a specific item is in the set

□ Dictionary

- map from keys to values, is a look-up table
e.g. telephone dictionary, amino acid letters to hydrophobicity values
 - need to get quickly the value for a key
-

1,2,3... action

how to do something

□ **statement**

- executes some function or operation, e.g.
`print 1+2`

□ **condition**

- describes when something is done, e.g.
`if number > 3:`
`print "greater than 3"`

□ **iteration**

- describes how to repeat something, e.g.
`for number in [1,2,3]:`
`print number`
-

condition

```
if condition :  
    do_something
```

```
if condition :  
    do_something  
else:  
    do_something_else
```

```
if condition :  
    do_something  
elif condition2 :  
    do_something_else1  
else:  
    do_something_else2
```

```
if x < 10:  
    print "in range"
```

```
if x < 5:  
    print "lower range"  
else:  
    print "out of range"
```

```
if x < 5:  
    print "lower range"  
elif x < 10:  
    print "upper range"  
else:  
    print "out of range"
```

iteration

```
for variable in sequence :  
    do_something
```

```
for color in ["red","green","blue"]:  
    print color
```

```
for i in xrange(10):  
    print i
```

```
for char in "some text":  
    print char
```

```
while condition :  
    statement1  
    statement2  
    ...
```

```
i = 0  
while i < 10 :  
    print i  
    i += 1
```

functions

- ❑ break complex problems in manageable pieces
- ❑ encapsulate/generalize common functionalities

```
def function(p1,p2,...):  
    do_something  
    return ...
```

```
def add(a,b):  
    return a+b
```

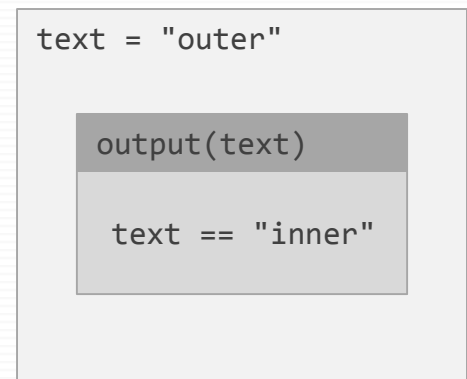
```
def divider():  
    print "-----"
```

```
def divider(ch,n):  
    print ch*n
```

```
def output(text):  
    print text
```

```
text = "outer"  
print text  
output("inner")  
print text
```

Scope of a variable



complete example

```
def count_gc(sequence):
    """Counts the nitrogenous bases of the given sequence.
    Ambiguous bases are counted fractionally.
    Sequence must be in upper case"""
    gc = 0
    for base in sequence:
        if base in 'GC':      gc += 1.0
        elif base in 'YRWSKM': gc += 0.5
        elif base in 'DH':    gc += 0.33
        elif base in 'VB':    gc += 0.66
    return gc

def gc_content(sequence):
    """Calculates the GC content of a DNA sequence.
    Mixed case, gaps and ambiguity codes are permitted"""
    sequence = sequence.upper().remove('-')
    if not sequence:
        return 0
    return 100.0 * count_gc(sequence) / len(sequence)

print gc_content("actacgattagag")
```

tips

1. no tabs, use 4 spaces for indentation
 2. lines should not be longer than 80 characters
 3. break complex code into small functions
 4. do not duplicate code, create functions instead
-

questions



Python Basics



let's play

- load fasta sequences
 - print name, length, first 10 symbols
 - min, max, mean length
 - find shortest
 - plot lengths histogram
 - calc GC content
 - write GCs to file
 - plot GC histogram
 - calc correlation coefficient
 - scatter plot
 - scatter plot over many
-

survival kit

IDLE:

Python doc: `F1`

Auto completion: `CTRL+SPACE/TAB`

Call tips: `CTRL+BACKSLASH`

History previous: `ALT+p`

History next: `ALT+n`

<http://www.quuux.com/stefan/slides.html>
<http://www.python.org>
<http://www.java2s.com/Code/Python/CatalogPython.htm>
<http://biopython.org>
<http://matplotlib.sourceforge.net/>
<http://www.scipy.org/Cookbook/Matplotlib>
<http://cheeseshop.python.org>
<http://www.scipy.org/Cookbook>

`help(...)`, `dir(...)`, google

Indentation has meaning!
Always 4 spaces, never tabs!

`def/if/for/while ... :`

```
#!/usr/bin/env python
$ chmod +x myscript.py
```

data types

```
# simple types
```

```
string : "string"
```

```
integer : 42
```

```
long : 4200000L
```

```
float : 3.145
```

```
hex : 0xFF
```

```
boolean : True
```

```
complex =:1+2j
```

```
# structured types
```

```
list = [1,2,'a']
```

```
tuple = (1,2,'a')
```

```
dict = {"pi":3.14, "e":2.17}
```

```
set([1,2,'a'])
```

```
frozenset([1,2,'a'])
```

```
func = lambda x,y: x+y
```

```
dir(3.14)
```

```
dir(float)
```

```
help(float)
```

```
help(float.__add__)
```

```
help(string)
```

all data types are objects

tuples

```
(1,2,3)
('red','green','blue')
('red',)
(1,) != (1)
()          # empty tuple
```

```
(1,2,3,4)[0]      # -> 1
(1,2,3,4)[2]      # -> 3
(1,2,3,4)[1:3]    # -> (2,3)
```

```
(a,b,c) = (1,2,3)
(a,b,c) = 1,2,3
a,b,c   = (1,2,3)
a,b,c   = [1,2,3]
```

```
a,b = b,a  # swap
```

tuples are not just round brackets

tuples are immutable

```
help(())
dir(())
```

```
for i,c in [(1,'I'), (2,'II'), (3,'III')]:
    print i,c
```

```
# vector addition
```

```
def add(v1, v2):
    x,y = v1[0]+v2[0], v1[1]+v2[1]
    return (x,y)
```

lists

```
[] == list()
nums = [1,2,3,4]
nums[0]
nums[:]
nums[0:2]
nums[:2]
nums[1:2]
nums[1:2] = 0
nums.append(5)
nums + [5,6]
```

```
range(5)
sum(nums)
max(nums)
```

```
[0]*5
```

```
nums.reverse()           # in place
nums2 = reversed(nums)  # new list
```

```
nums.sort()              # in place
nums2 = sorted(nums)     # new list
```

lists are mutable*

lists are arrays

*since lists are mutable you cannot use them as a dictionary keys!

```
dir([]), dir(list)
help([])
help([].sort)
```

lists examples

```
l = [('a',3), ('b',2), ('c',1)]
l.sort(key = lambda x: x[1])
l.sort(key = lambda (c,n): n)
l.sort(cmp = lambda x,y: x[1]-y[1])
l.sort(cmp = lambda (c1,n1),(c2,n2): n1-n2)
```

```
colors = ['red','green','blue']
colstr = ''
for color in colors:
    colstr = colstr+', '+color
colstr = ",".join(colors)
```

```
l1 = ['a','b','c']
l2 = [1,2,3]
l3 = zip(l1,l2)
zip(*l3)
```

```
mat = [(1,2,3),
        (4,5,6)]
flip = zip(*mat)
flipback = zip(*flip)
```


slicing

```
s = "another string"
```

```
s[0:len(s)]
```

```
s[:]
```

```
s[2:7]
```

```
s[-1]
```

```
s[:-1]
```

```
s[-6:]
```

```
s[:-6]
```

```
s[::2]
```

```
s[:: -1]
```

```
from numpy import array
mat = array([[1,2,3],
             [4,5,6]])
```

```
mat[1][1]
```

```
mat[:,:]
```

```
mat[1:3, 0:2]
```

```
mat[1:3, ...]
```

`slice[start:end:stride]`

start **inclusive**

end **exclusive**

stride **optional**

slicing works the same for lists and tuples (<= sequences)

sets

```
set([3,2,2,3,4])  
frozenset([3,2,2,3,4])
```

```
s = "my little string"  
set(s)
```

```
s.remove('t')  
s.pop()
```

```
s1 = set([1,2,3,4])  
s2 = set([3,4,5])  
s1.union(s2)  
s1.difference(s2)  
s1 - s2  
s1 or s2  
s1 and s2
```

sets are mutable

frozensets are immutable

```
dir(set())  
help(set)  
help(set.add)  
help(frozenset)
```

```
s = set([2,3,3,34,51,1])  
max(s)  
min(s)  
sum(s)
```

dictionaries

```
d = {}  
d = dict()  
d = {'pi':3.14, 'e':2.7}  
d = dict(pi=3.14, e=2.7)  
d = dict([('pi',3.14),('e',2.7)])
```

directories are hashes

only immutables are allowed
as keys

```
d['pi']  
d['pi'] = 3.0  
d['zero'] = 0.0  
d[math.pi] = "pi"  
d[(1,2)] = "one and two"
```

```
dir({})  
help({})  
help(dict)  
help(dict.values)  
help(dict.keys)
```

```
d.get('two', 2)  
d.setdefault('one', 1)  
d.has_key('one')  
'one' in d
```

```
mat = [[0,1], [1,3], [2,0]]  
sparse = dict([(i,j),e]  
               for i,r in enumerate(mat)  
               for j,e in enumerate(r) if e])
```

data structures - tips

when to use what?

- ❑ **List**
 - many similar items to store
e.g, numbers, protein ids, sequences, ...
 - no need to **find** a specific item fast
 - fast access to items at a specific **position** in the list
 - ❑ **Tuple**
 - a few (<10), different items to store
e.g. addresses, protein id and its sequence, ...
 - want to use it as dictionary key
 - ❑ **Set**
 - many, unique items to store
e.g. unique protein ids
 - need to know quickly if a specific item is in the set
 - ❑ **Dictionary**
 - map from keys to values, is a look-up table
e.g. telephone dictionary, amino acid letters to hydrophobicity values
 - need to get quickly the value for a key
-

boolean logic

False: False, 0, None, [], (,)

True: everything else, e.g.: 1, True, ['blah'], ...

A = 1

B = 2

A and B

A or B

not A

1 in [1,2,3]

"b" in "abc"

all([1,1,1])

any([0,1,0])

l1 = [1,2,3]

l2 = [4,5]

if not l1:

print "list is empty or None"

if l1 and l2:

print "both lists are filled"

comparisons

```
(1, 2, 3) < (1, 2, 4)
[1, 2, 3] < [1, 2, 4]
'C' < 'Pascal' < 'Perl' < 'Python'
(1, 2, 3, 4) < (1, 2, 4)
(1, 2) < (1, 2, -1)
(1, 2, 3) == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'), 4)
```

comparison of complex objects
chained comparisons

```
s1 = "string1"
s2 = "string2"
s1 = s3
```

```
s1 == s2    # same content
s1 is s3    # same reference
```

} Java people watch out!

if

```
if 1 < x < 10:  
    print "in range"
```

indentation has meaning
there is no switch() statement

```
if 1 < x < 5:  
    print "lower range"  
elif 5 < x < 10:  
    print "upper range"  
else:  
    print "out of range"
```

```
if condition :  
    do_something  
elif condition2 :  
    do_something_else1  
else:  
    do_something_else2
```

```
# conditional expression  
frac = 1/x if x>0 else 0
```

```
# one-line if  
if condition : statement
```

```
result = statement if condition else alternative
```

for

```
for i in xrange(10):  
    print i
```

```
for i in xrange(10,0,-1):  
    print i
```

```
for ch in "mystring":  
    print ch
```

```
for e in ["red","green","blue"]:  
    print e
```

```
for line in open("myfile.txt"):  
    print line
```

```
for variable in sequence :  
    statement1  
    statement2  
    ...
```

```
help(range)  
help(xrange)
```

more for

```
for i in xrange(10):  
    if 2<i<5 :  
        continue  
    print i
```

```
for ch in "this is a string":  
    if ch == ' ':  
        break  
    print ch
```

```
i = 0  
for ch in "mystring":  
    print i,ch  
    i = i + 1
```

```
nums = [1,2,3,4,5]  
for i in nums:  
    if i==3: del nums[3]  
    print i
```

```
for i,ch in enumerate("mystring"):  
    print i,ch
```

```
for i,line in enumerate(open("myfile.txt")):  
    print i,line
```

Don't modify list
while iterating
over it!

while

```
i = 0
while i < 10 :
    print i
    i += 1
```

```
while condition :
    statement1
    statement2
...
```

```
i = 0
while 1 :
    print i
    i += 1
    if i >= 10:
        break
```

```
i = 0
while 1 :
    i += 1
    if i < 5:
        continue
    print i
```

strings

`"quotes"`

`'apostrophes'`

`'You can "mix" them'`

`'or you \'escape\' them'`

`"a tab \t and a newline \n"`

`"""Text over
multiple lines"""`

`'''Or like this,
if you like.'''`

strings are immutable

`r"(a-z)+\.doc"`

`# äöü`

`u"\xe4\xfc\xfc"`

`"a"+" "+string"`

`"repeat "*3`

if you code in C/C++/Java/... as well, I suggest apostrophes for characters and quotes for strings, e.g: `'c'` and `"string"`

string formatting

```
print "new line"  
print "same line",
```

```
"height=",12," meters"  
"height="+str(12)+" meters"  
"height=%d meters" % 12  
"%s=%.3f meters or %d cm" % ("height", 1.0, 100)
```

```
# template strings  
dic = {"prop1":"height", "len":100, "color":"green"}  
"%(prop1)s = %(len)d cm" % dic  
"The color is %(color)s" % dic
```

format codes (%d, %s, %f, ...) similar to C/C++/Java

string methods

```
s = " my little string "
```

```
len(s)
```

```
s.find("string")
```

```
s.count("t")
```

```
s.strip()
```

```
s.replace("my", "your")
```

```
s[4]
```

```
s[4:10]
```

```
":".join(["red", "green", "blue"])
```

```
str(3.14); float("3.14"); int("3")
```

```
dir("")
```

```
dir(str)
```

```
help("").count)
```

```
help(str)
```

references

```
v1 = 10
v2 = v1      -> v2 = 10    # content copied
v1 = 50      -> v2 = 10    # as expected
```

```
import copy
help(copy.copy)
help(copy.deepcopy)
```

```
l1 = [10]
l2 = l1      -> l2 = [10] # address copied
l1[0] = 50   -> l2 = [50] # oops
```

```
l1 = [10]
l2 = l1[:]   -> l2 = [10] # content copied
l1[0] = 50   -> l2 = [10] # that's okay now
```

same for sets (and dictionaries) but not for tuples, strings or frozensets (<- immutable)

list comprehension

[expression for variable in sequence if condition]

condition is optional

```
[x*x for x in xrange(10)]           # square
```

```
[x for x in xrange(10) if not x%2]  # even numbers
```

```
[(b,a) for a,b in [(1,2), (3,4)]]   # swap
```

```
s = "mary has a little lamb"  
[ord(c) for c in s]  
[i for i,c in enumerate(s) if c==' ']
```

```
# what's this doing?
```

```
[p for p in xrange(100) if not [x for x in xrange(2,p) if not p%x]]
```

generators

(expression for variable in sequence if condition)

```
(x*x for x in xrange(10))
```

```
for n in (x*x for x in xrange(10)):
    print n
```

```
sum(x*x for x in xrange(10))
```

```
"-".join(c for c in "try this")
```

```
def xrange1(n):
    return (x+1 for x in xrange(n))
```

```
def my_xrange(n):
    i = 0
    while i < n :
        i += 1
        yield i
```

```
def my_range(n):
    l = []
    i = 0
    while i < n :
        i += 1
        l.append(i)
    return l
```


functions

```
def add(a, b):  
    return a+b
```

```
def inc(a, b=1):  
    return a+b
```

help(add)

```
def add(a, b):  
    """adds a and b"""  
    return a+b
```

```
def list_add(l1):  
    return sum(l1)
```

```
def list_add(l1, l2):  
    return [a+b for a,b in zip(l1,l2)]
```

```
def function(p1,p2,...):  
    """ doc string """  
    ...  
    return ...
```

```
# duck typing  
add(1,2)  
add("my", "string")  
add([1, 2], [3, 4])
```

functions - args

```
def function(p1,p2,...,*args,*kwargs):  
    return ...
```

```
def add(*args):  
    """example: add(1,2,3)"""  
    return sum(args)
```

variable arguments are lists
or dictionaries

```
def scaled_add(c, *args):  
    """example: scaled_add(2, 1,2,3)"""  
    return c*sum(args)
```

```
def showme(*args):  
    print args
```

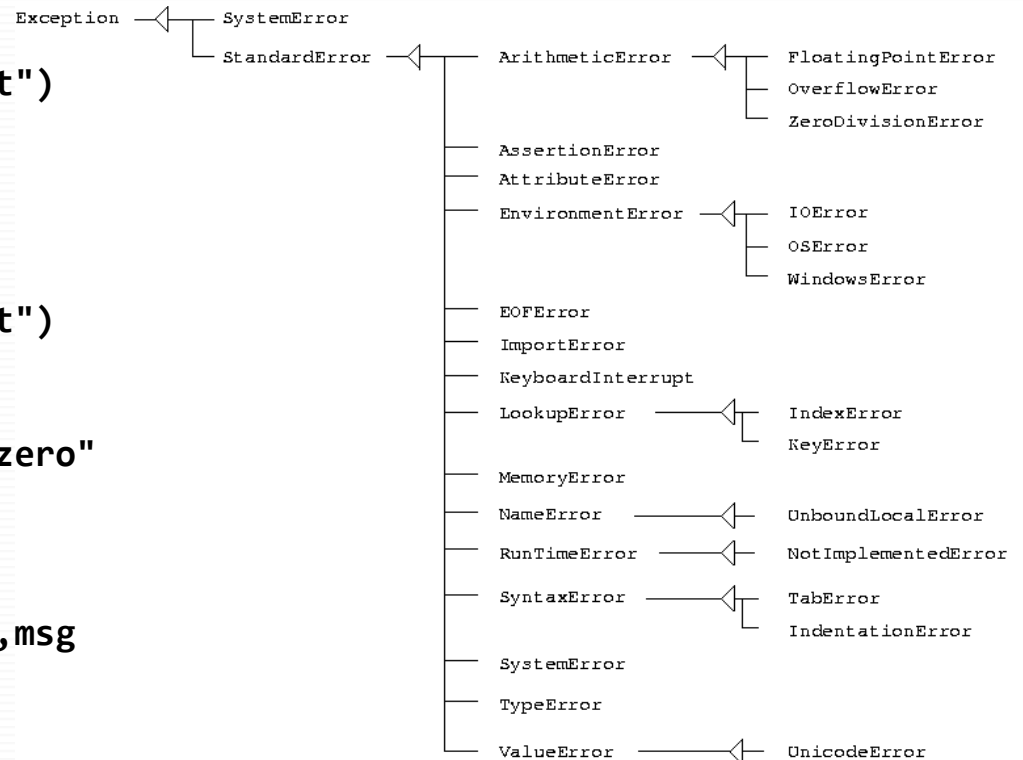
```
def super_add(*args, **kwargs):  
    """example: super_add(1,2,3, scale=2)"""  
    scale = kwargs.get('scale', 1)  
    offset = kwargs.get('offset', 0)  
    return offset + scale * sum(args)
```

```
def showmemore(**kwargs):  
    print kwargs
```

Exceptions - handle

```
try:
    f = open("c:/somefile.txt")
except:
    print "cannot open file"

try:
    f = open("c:/somefile.txt")
    x = 1/y
except ZeroDivisionError:
    print "cannot divide by zero"
except IOError, msg:
    print "file error: ",msg
except Exception, msg:
    print "ouch, surprise: ",msg
else:
    x = x+1
finally:
    f.close()
```



Exceptions - raise

```
try:
    # do something and raise an exception
    raise IOError, "Something went wrong"
except IOError, error_text:
    print error_text
```

doctest

```
def add(a, b):  
    """Adds two numbers or lists  
    >>> add(1,2)  
    3  
    >>> add([1,2], [3,4])  
    [1, 2, 3, 4]  
    """  
    return a+b
```

```
if __name__ == "__main__":  
    import doctest  
    doctest.testmod()
```

unittest

```
import unittest

def add(a,b): return a+b
def mult(a,b): return a*b

class TestCalculator(unittest.TestCase):
    def test_add(self):
        self.assertEqual( 4, add(1,3))
        self.assertEqual( 0, add(0,0))
        self.assertEqual(-3, add(-1,-2))
    def test_mult(self):
        self.assertEqual( 3, mult(1,3))
        self.assertEqual( 0, mult(0,3))

if __name__ == "__main__":
    unittest.main()
```

import

```
import math
math.sin(3.14)
```

```
from math import sin
sin(3.14)
math.cos(3.14)
```

```
from math import sin, cos
sin(3.14)
cos(3.14)
```

```
from math import *      # careful!
sin(3.14)
cos(3.14)
```

```
import math as m
m.sin(3.14)
m.cos(m.pi)
```

```
import module
import module as m
from module import f1, f2
from module import *
```

```
import math
help(math)
dir(math)
help(math.sin)
```

import example

```
# module calculator.py

def add(a,b):
    return a+b

if __name__ == "__main__":
    print add(1,2)    # test
```

```
# module do_calcs.py

import calculator

def main():
    print calculator.add(3,4)

if __name__ == "__main__":
    main()
```


package example

```
calcpack/__init__.py  
calcpack/calculator.py  
calcpack/do_calcs.py
```

```
# in a different package  
  
from calcpack.calculator import add  
x = add(1,2)  
  
from calcpack.do_calcs import main  
main()
```

template

```
""" This module implements  
some calculator functions  
"""
```

```
def add(a,b):  
    """Adds two numbers  
    a -- first number  
    b -- second number  
    returns the sum """  
    return a+b
```

```
def main():  
    """Main method. Adds 1 and 2"""  
    print add(1,2)
```

```
if __name__ == "__main__":  
    main()
```

regular expressions

```
import re
```

```
text = "date is 24/07/2008"  
re.findall(r'(..)/(..)/(....)', text)  
re.split(r'[\s/]', text)  
re.match(r'date is (.*)', text).group(1)  
re.sub(r'(../)(../)', r'\2\1', text)
```

```
# compile pattern if used multiple times  
pattern = compile(r'(..)/(..)/(....)')  
pattern.findall(text)  
pattern.split(...)  
pattern.match(...)  
pattern.sub(...)
```

Perl addicts:

only use regex if there is
no other way.

Tip: string methods and
data structures

file reading/writing

```
f = open(fname)
for line in f:
    print line
f.close()
```

```
open(fname).read()
open(fname).readline()
open(fname).readlines()
```

```
dir(file)
help(file)
```

```
f = open(fname, 'w')
f.write("blah blah")
f.close()
```

```
#skip header and first col
f = open(fname)
f.next()
for line in f:
    print line[1:]
f.close()
```

```
def write_matrix(fname, mat):
    f = open(fname, 'w')
    f.writelines([' ' .join(map(str, row))+'\n' for row in mat])
    f.close()
```

```
def read_matrix(fname):
    return [map(float, line.split()) for line in open(fname)]
```

file handling

```
import os.path as path
path.split("c:/myfolder/test.dat")
path.join("c:/myfolder", "test.dat")
```

```
import os
os.listdir('.')
os.getcwd()
```

```
import glob
glob.glob("*.py")
```

```
import os
dir(os)
help(os.walk)
```

```
import os.path
dir(os.path)
```

```
import shutil
dir(shutil)
help(shutil.move)
```

file processing examples

```
def number_of_lines(fname):  
    return len(open(fname).readlines())
```

```
def number_of_words(fname):  
    return len(open(fname).read().split())
```

```
def enumerate_lines(fname):  
    return [t for t in enumerate(open(fname))]
```

```
def shortest_line(fname):  
    return min(enumerate(open(fname)), key=lambda (i,l): len(l))
```

```
def wordiest_line(fname):  
    return max(enumerate(open(fname)), key=lambda (i,l): len(l.split()))
```

system

```
import sys
if __name__ == "__main__":
    args = sys.argv
    print "script name: ", args[1]
    print "script args: ", args[1:]
```

```
import os
# run and wait
os.system("mydir/blast -o %s" % fname)
```

```
import subprocess
# run and do not wait
subprocess.Popen("mydir/blast -o %s" % fname, shell=True)
```

```
import sys
dir(sys)
sys.version
sys.path
```

```
import os
dir(os)
```

```
help(os.sys)
help(os.getcwd)
help(os.mkdir)
```

last famous words

1. line length < 80
 2. complexity < 10
 3. no code duplication
 4. value-adding comments
 5. use language idioms
 6. automated tests
-

questions



Advanced Python



overview

- Functional programming
- Object oriented programming

Functional programming

Functional Programming is a programming paradigm that emphasizes the application of functions and avoids state and mutable data, in contrast to the imperative programming style, which emphasizes changes in state.

- ❑ makes some things easier
- ❑ limited support in Python

Functions can be treated like any other type of data

```
def timeFormat(date):  
    return "%2d:%2d" % (date.hour, date.min)
```

```
def dayFormat(date):  
    return "Day: %sd" % (date.day)
```

```
def datePrinter(dates, format):  
    for date in dates:  
        print format(date)
```

```
datePrinter(dates, timeFormat)
```


```
def add(a,b):  
    return a+b  
plus = add  
plus(1,2)
```

```
def inc_factory(n):  
    def inc(a):  
        return n+a  
    return inc  
inc2 = inc_factory(2)  
inc3 = inc_factory(3)  
inc3(7)
```

FP - lambda functions

Lambda functions are **anonymous** functions.
Typically for **very short functions** that are **used only once**.

`l = [('a',3), ('b',2), ('c',1)]`



```
#without lambda functions
def key(x): return x[1]
l.sort(key = key)

def cmp(x,y): return x[1]-y[1]
l.sort(cmp = cmp)
```

```
#with lambda functions
l.sort(key = lambda (c,n): n)
l.sort(cmp = lambda x,y: x[1]-y[1])
```

functional programming

map applies a function to the elements of a sequence

```
map(str, [1,2,3,4,5])
```

```
l = []  
for n in [1,2,3,4,5]:  
    l.append(str(n))
```

filter extracts elements from a sequence depending on a predicate function

```
filter(lambda x: x>3, [1,2,3,4,5])
```

```
l = []  
for x in [1,2,3,4,5]:  
    if x>3: l.append(n)
```

reduce iteratively applies a binary function, reducing a sequence to a single element


```
reduce(lambda a,b: a*b, [1,2,3,4,5])
```

```
prod = 1  
for x in [1,2,3,4,5]:  
    prod = prod * x
```

FP - example

Problem

sum over matrix rows stored in a file

File		List
1 2 3		6
4 5 6		15

Imperative

```
rowsums = []
for row in open(fname):
    elems = row.split()
    rowsum = 0
    for e in elems:
        rowsum += float(e)
    rowsums.append(rowsum)
```

Functional

```
rowsums = [sum(map(float,row.split())) for row in open(fname)]
```

Extra Functional ;-)

```
rowsums = map(sum,map(lambda row:map(float,row.split()),open(fname)))
```

Numpy

```
rowsums = sum(loadtxt(fname),axis=1)
```

FP - more examples

```
numbers = [1,2,3,4]
numstr  = ",".join(map(str,numbers))
numbers = map(int,numstr.split(','))
```

```
v1 = [1,2,3]
v2 = [3,4,5]
dotprod = sum(map(lambda (x,y): x*y, zip(v1,v2)))
dotprod = sum(x*y for x,y in zip(v1,v2))
```


Object Oriented Programming

Object-oriented programming (OOP) is a programming paradigm that uses "objects"
– data structures consisting of data fields and associated methods.

- ❑ brings data and functions together
- ❑ helps to manage complex code
- ❑ limited support in Python

Dot notation

`object.attribute`
`object.method()`

Class

- attributes
- + methods

Car

- color
- brand
- + consumption(speed)

Examples

```
text = "some text"
text.upper()
len(text) #not a method call
text.__len__()
```

```
f = open(filename)
if not f.closed :
    lines = f.readlines()
f.close()
```

```
try:
    dir(file)
    help(file)
```

OO motivation

Problem

for some genes print out name, length and GC content

Non-OO approach

```
genes = [  
    ['gatA', 2108, 3583, 'agaccta'],  
    ['yfgA', 9373, 9804, 'agaaa'],  
    ...  
]  
  
def gc_content(seq):  
    ...  
    return gc
```

```
for gene in genes:  
    print gene[0], gene[2]-gene[1], gc_content(gene[3])
```

Object oriented

```
for gene in genes:  
    print gene.name(), gene.length(), gene.gc_content()
```

OO definitions

Class: a template that defines attributes and functions of something,
e.g.

Car

- brand
- color
- calc_fuel_consumption(speed)

Attributes, Fields, Properties: things that describe an object,
e.g. Brand, Color

Methods, Operations, Functions: something that an object can do
e.g. calc_fuel_consumption(speed)

Instance: an actual, specific object created from the template/class
e.g. red BMW M5

Object: some unspecified class instance

Decorators

- Minor Syntax tweak but...
 - Syntax matters, often in unexpected ways
- they modify functions, and in the case of *class decorators*, entire classes

```
@foo
```

```
def bar: pass
```

Same as

```
bar = foo(bar)
```

```
def foo(): pass
```

```
foo = staticmethod(foo)
```

```
@staticmethod
```

```
def foo(): pass
```

Decorators

- Function based decorators

```
def trace(f):  
    def new_f(*args)  
        print 'Entering %s%s' % (f.__name__, args)  
        result = f(*args, **kwargs)  
        print 'Exiting %s%s with %s' % (f.__name__, args, result)  
        return result  
    return new_f
```

@trace

```
def sum(n, m):  
    return n + m
```

Further Reading

- <http://wiki.python.org/moin/PythonDecoratorLibrary>
 - More examples of decorators. Note the number of these examples that use classes rather than functions as decorators.
- <http://scratch.tplus1.com/decoratortalk>
 - Matt Wilson's *Decorators* [see this](#)

Metaprogramming

- Every class statement uses a metaclass
 - Mostly type (or `types.ClassType`)
 - No hassle, no problem, no issue
- However, you can make a `custom` metaclass

```
class SimpleMeta1(type):  
    def __init__(cls, name, bases, nmspc):  
        super(SimpleMeta1, cls).__init__(name, bases, nmspc)  
        cls.uses_metaclass = lambda self : "Yes!"
```

```
class Simple1(object):  
    __metaclass__ = SimpleMeta1  
    def foo(self): pass  
    @staticmethod  
    def bar(): pass
```

```
simple = Simple1()
```

Metaprogramming

- Metaprogramming involves hooking our own operations into the creation of class objects
 - `__init__`
 - `__new__`
 - `__call__`
 - **Further Reading**
 - Excellent step-by-step introduction to metaclasses:
<http://cleverdevil.org/computing/78/>
 - Metaclass intro and comparison of syntax between Python 2.x and 3.x: <http://mikewatkins.ca/2008/11/29/python-2-and-3-metaclasses/>
-

The Pattern Concept

- “Design patterns help you learn from others’ successes instead of your own failures”
 - Creational
 - Structural
 - Behavioral
 - **Further Reading**
 - Alex Martelli’s Video Lectures on Design Patterns in Python:
<http://www.catonmat.net/blog/learning-python-design-patterns-through-video-lectures/>
-

Unit Testing & Test-Driven Development

- TBD

questions



links

- Wikipedia – Python
<http://en.wikipedia.org/wiki/Python>
 - Learn Python Hard way
<http://learnpythonthehardway.org/>
 - Python Beginner
<https://www.python.org/about/gettingstarted/>
 - Instant Python
<http://hetland.org/writing/instant-python.html>
 - Dive into Python
<http://www.diveintopython.org/>
 - Hitchhiker Guide to Python
<http://docs.python-guide.org/en/latest/intro/learning/>
 - Python Magic Method
<http://www.rafekettler.com/magicmethods.html>
-

books



Learn Python the Hard Way
By Zed A. Shaw
2009

Expert Python Programming
By Tarek Ziade
2008

Thank You

Design is not what it look like & feels like.

Design is how it works.

- Steve Jobs