Conditionals, Looping, Sequences and Lists

Christopher Barker

IRIS

October 16, 2013

Table of Contents

Review

Questions from yesterday?

Recursion: python supports it. See:
`Session1/code/ackerman.rst`

Truthiness

What is true or false in Python?

- The Booleans: `True` and `False`
- "Something or Nothing"

`http://mail.python.org/pipermail/python-dev/2002-April/022107.html`

Truthiness

Determining Truthiness:

```
bool(something)
```

Boolean Expressions

`False`

- `None`
- `False`
- zero of any numeric type, for example, `0, 0L, 0.0, 0j`.
- any empty sequence, for example, `'', (), []` .
- any empty mapping, for example, `{}`.
- instances of user-defined classes, if the class defines a `__nonzero__()` or `__len__()` method, when that method returns the integer zero or bool value `False`.

`http://docs.python.org/library/stdtypes.html`

Boolean Expressions

Avoid:

```
if xx == True:
```

Use:

```
if xx:
```

Boolean Expressions

"Shortcutting"

```
                        if x is false,
x or y                     return y,
                           else return x

                        if x is false,
x and y                     return  x
                            else return y

                        if x is false,
not x                     return True,
                          else return False
```

Boolean Expressions

Stringing them together

```
a or b or c or d
```

```
a and b and c and d
```

The first value that defines the result is returned

(demo)

Boolean returns

From CodingBat

(http://codingbat.com/python)

```
def makes10(a, b):
    """Given 2 ints, a and b, return True if one if them is
       or if their sum is 10. """
    return a == 10 or b == 10 or a+b == 10
```

http://codingbat.com/prob/p173401

```
def sleep_in(weekday, vacation):
    return not (weekday == True and vacation == False)
```

or

```
def sleep_in(weekday, vacation):
    return (not weekday) or vacation
```

bools are ints?

bool types are subclasses of integer

```
In [1]: True == 1
Out[1]: True

In [2]: False == 0
Out[2]: True
```

It gets weirder!

```
In [6]: 3 + True
Out[6]: 4
```

(demo)

Conditional expression

A common idiom:

```
if something:
    x = a_value
else:
    x = another_value
```

Also, other languages have a "ternary operator"
(C family: `result = a > b ? x : y ;`)

```
y = 5 if x > 2 else 3
```

PEP 308: (http://www.python.org/dev/peps/pep-0308/)

LAB

- Look up the % operator. What do these do?

```
10 % 7 == 3
14 % 7 == 0
```

- Write a program that prints the numbers from 1 to 100 inclusive. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz" instead.
- Re-write a couple CodingBat exercises, using a conditional expression
- Re-write a couple CodingBat exercises, returning the direct boolean results

(use whichever you like, or the ones in: `code/codingbat.rst`)

Code Structure

Python is all about namespaces – the "dots"

`name.another_name`

The "dot" indicates looking for a name in the namespace of the given object. It could be:

- name in a module
- module in a package
- attribute of an object
- method of an object

indenting and blocks

Indenting determines blocks of code

```
something:
    some code
    some more code
    another block:
        code in
        that block
```

But you need the colon too...

indenting and blocks

You can put a one-liner after the colon:

```
In [167]: x = 12

In [168]: if x > 4: print x
12
```

Only do this if it makes it more readable...

Spaces and Tabs

An indent can be:

- Any number of spaces
- A tab
- tabs and spaces:
  - A tab is eight spaces (always!)
  - Are they eight in your editor?

Always use four spaces – really!

(PEP 8)

Spaces Elsewhere

Other than indenting – space doesn't matter

```
x = 3*4+12/func(x,y,z)
```

```
x = 3*4 + 12 /   func (x,   y, z)
```

Choose based on readability/coding style

PEP 8

Various Brackets

Bracket types:

- parentheses `( )`
  - tuple literal: `(1,2,3)`
  - function call: `fun( arg1, arg2 )`
  - grouping: `(a + b) * c`
- square brackets `[ ]`
  - list literal: `[1,2,3]`
  - sequence indexing: `a_string[4]`
- curly brackets `{ }`
  - dictionary literal: `{"this":3, "that":6}`
  - (we'll get to those...)

modules and packages

A module is simply a namespace

A package is a module with other modules in it

The code in the module is run when it is imported

importing modules

```
import modulename

from modulename import this, that

import modulename as a_new_name
```

(demo)

importing from packages

```
import packagename.modulename

from packagename.modulename import this, that

from package import modulename
```

(demo)

http://effbot.org/zone/import-confusion.htm

importing from packages

```
from modulename import *
```

Don't do this!

("Namespaces are one honking great idea...")

(wxPython and numpy example...)

Except *maybe* math module

(demo)

import

If you dont know the module name before execution.

`__import__(module)`

where `module` is a Python string.

modules and packages

The code in a module is NOT re-run when imported again – it must be explicitly reloaded to be re-run

```
import modulename
```

```
reload(modulename)
```

(demo)

```
import sys
print sys.modules
```

(demo)

LAB

Experiment with importing different ways:

```
import math
dir(math) # or, in ipython -- math.<tab>
math.sqrt(4)

import math as m
m.sqrt(4)

from math import *
sqrt(4)
```

LAB

Experiment with importing different ways:

```
import sys
print sys.path

import os
print os.path
```

You wouldn't want to import * those – check out

```
os.path.split()
os.path.join()
```

Sequences

Sequences are ordered collections of objects

They can be indexed, sliced, iterated over,...

They have a length: `len(sequence)`

Common sequences (Remember Duck Typing?):

- strings
- tuples
- lists

Indexing

square brackets for indexing: []

Indexing starts at zero

```
In [98]: s = "this is a string"

In [99]: s[0]
Out[99]: 't'

In [100]: s[5]
Out[100]: 'i'
```

Indexing

Negative indexes count from the end

```
In [105]: s = "this is a string"

In [106]: s[-1]
Out[106]: 'g'

In [107]: s[-6]
Out[107]: 's'
```

Slices

Slicing: Pulling a range out of a sequence

```
sequence[start:finish]
```

```
indexes for which:
```

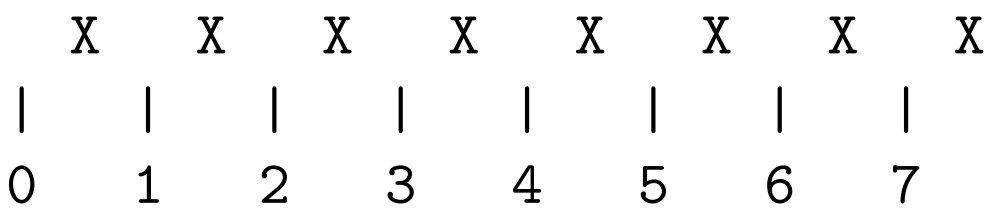
```
start <= i < finish
```

Slices

```
In [121]: s = "a bunch of words"
In [122]: s[2]
Out[122]: 'b'

In [123]: s[6]
Out[123]: 'h'

In [124]: s[2:6]
Out[124]: 'bunc'

In [125]: s[2:7]
Out[125]: 'bunch'
```

Slices

the indexes point to the spaces between the items



Slices

Slicing satisfies nifty properties:

```
len( seq[a:b] ) == b - a
```

```
seq[a:b] + seq[b:c] == seq
```

Slicing vs. Indexing

Indexing returns a single element

```
In [86]: l
Out[86]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [87]: type(l)
Out[87]: list

In [88]: l[3]
Out[88]: 3

In [89]: type( l[3] )
Out[89]: int
```

Slicing vs. Indexing

Unless it's a string:

```
In [75]: s = "a string"

In [76]: s[3]
Out[76]: 't'

In [77]: type(s[3])
Out[77]: str
```

There is no single character type

Slicing vs. Indexing

Slicing returns a sequence:

```
In [68]: l
Out[68]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [69]: l[2:4]
Out[69]: [2, 3]
```

Even if it's one element long

```
In [70]: l[2:3]
Out[70]: [2]

In [71]: type(l[2:3])
Out[71]: list
```

Slicing vs. Indexing

Indexing out of range produces an error

```
In [129]: s = "a bunch of words"
In [130]: s[17]
----> 1 s[17]
IndexError: string index out of range
```

Slicing just gives you what's there

```
In [131]: s[10:20]
Out[131]: ' words'

In [132]: s[20:30]
Out[132]: ''
```

(demo)

Multiplying and slicing

from CodingBat: Warmup-1 – front3
(http://codingbat.com/prob/p147920)

```
def front3(str):
  if len(str) < 3:
    return str+str+str
  else:
    return str[:3]+str[:3]+str[:3]
```

or

```
def front3(str):
    return str[:3] * 3
```

Slicing

from CodingBat: Warmup-1 – `missing_char`
(`http://codingbat.com/prob/p149524`)

```
def missing_char(str, n):
  front = str[0:n]
  l = len(str)-1
  back = str[n+1:l+1]
  return front + back

def missing_char(str, n):
    return str[:n] + str[n+1:]
```

Slicing

you can skip items, too

```
In [289]: string = "a fairly long string"

In [290]: string[0:15]
Out[290]: 'a fairly long s'

In [291]: string[0:15:2]
Out[291]: 'afil ogs'

In [292]: string[0:15:3]
Out[292]: 'aallg'
```

LAB

Write some functions that:

- return a string with the first and last characters exchanged.
- return a string with every other character removed
- return a string with the first and last 4 characters removed, and every other char in between
- return a string reversed (just with slicing)
- return a string with the middle, then last, then first third in a new order

Lists

List Literals

```
>>> []
[]
>>> list()
[]
>>> [1, 2, 3]
[1, 2, 3]
>>> [1, 3.14, "abc"]
[1, 3.14, 'abc']
```

Lists

List "type"
(also constructor)

```
>>> type(list)
<type 'type'>
>>> list( (1,2,3) )
[1, 2, 3]
>>> list( "a string" )
```

Takes any sequence, tries to turn it into a list

like `int()`, `float()`, etc.

List Indexing

Indexing just like all sequences

```
>>> food = ['spam', 'eggs', 'ham']
>>> food[2]
'ham'
>>> food[0]
'spam'
>>> food[42]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

List Mutability

Lists are mutable

```
>>> food = ['spam', 'eggs', 'ham']
>>> food[1] = 'raspberries'
>>> food
['spam', 'raspberries', 'ham']
```

List Elements

Each element is a value, and can be in multiple lists and have multiple names (or no name)

```
>>> name = 'Brian'
 >>> a = [1, 2, name]
 >>> b = [3, 4, name]
 >>> name
 'Brian'
 >>> a
 [1, 2, 'Brian']
 >>> b
 [3, 4, 'Brian']
 >>> a[2]
 'Brian'
 >>> b[2]
```

List Methods

`.append()`, `.insert()`

```
>>> food = ['spam', 'eggs', 'ham']
>>> food.append('sushi')
>>> food
['spam', 'eggs', 'ham', 'sushi']
>>> food.insert(0, 'carrots')
>>> food
['carrots', 'spam', 'eggs', 'ham', 'sushi']
```

List Methods

.extend()

```
>>> food = ['spam', 'eggs', 'ham']
>>> food.extend(['fish', 'chips'])
>>> food
['spam', 'eggs', 'ham', 'fish', 'chips']
```

could be any sequence:

```
>>>  food
>>>  ['spam', 'eggs', 'ham']
>>>  silverware = ('fork', 'knife', 'spoon') # a tuple
>>>  food.extend(silverware)
>>>  food
>>>  ['spam', 'eggs', 'ham', 'fork', 'knife', 'spoon']
```

List Methods

pop(), remove()

```
In [203]: food = ['spam', 'eggs', 'ham', 'toast']
In [204]: food.pop()
Out[204]: 'toast'
In [205]: food.pop(0)
Out[205]: 'spam'
In [206]: food
Out[206]: ['eggs', 'ham']
In [207]: food.remove('ham')
In [208]: food
Out[208]: ['eggs']
```

List Constructor

`list()` accepts any sequence and returns a list of that sequence

```
>>> word = 'Python '
>>> chars = []
>>> for char in word:
...     chars.append(char)
>>> chars
['P', 'y', 't', 'h', 'o', 'n', ' ']
>>> list(word)
['P', 'y', 't', 'h', 'o', 'n', ' ']
```

List Slicing

Slicing makes a copy

```
In [227]: food = ['spam', 'eggs', 'ham', 'sushi']

In [228]: some_food = food[1:3]

In [229]: some_food[1] = 'bacon'

In [230]: food
Out[230]: ['spam', 'eggs', 'ham', 'sushi']

In [231]: some_food
Out[231]: ['eggs', 'bacon']
```

List Slicing

Easy way to copy a whole list

```
In [232]: food
Out[232]: ['spam', 'eggs', 'ham', 'sushi']

In [233]: food2 = food[:]

In [234]: food is food2
Out[234]: False
```

but the copy is "shallow":

http://docs.python.org/library/copy.html

List Slicing

"Shallow" copy

```
In [249]: food = ['spam', ['eggs', 'ham']]
In [251]: food_copy = food[:]
In [252]: food[1].pop()
Out[252]: 'ham'
In [253]: food
Out[253]: ['spam', ['eggs']]
In [256]: food.pop(0)
Out[256]: 'spam'
In [257]: food
Out[257]: [['eggs']]
In [258]: food_copy
Out[258]: ['spam', ['eggs']]
```

Name Binding

Assigning to a name does not copy:

```
>>> food = ['spam', 'eggs', 'ham', 'sushi']
>>> food_again = food
>>> food_copy = food[:]
>>> food.remove('sushi')
>>> food
['spam', 'eggs', 'ham']
>>> food_again
['spam', 'eggs', 'ham']
>>> food_copy
['spam', 'eggs', 'ham', 'sushi']
```

List Iterating

Iterating over a list

```
>>> food = ['spam', 'eggs', 'ham', 'sushi']
>>> for x in food:
...     print x
...
spam
eggs
ham
sushi
```

Processing Lists

A common pattern

```
filtered = []
for x in somelist:
    if should_be_included(x):
        filtered.append(x)
del(somelist)  # maybe
```

you don't want to be deleting items from the list while iterating...

Mutating Lists

if you're going to change the list, iterate over a copy for safety

```
>>> food = ['spam', 'eggs', 'ham', 'sushi']
>>> for x in food[:]:
    ...    # change the list somehow
    ...
```

insidious bugs otherwise

operators vs methods

What's the difference?

```
>>> food = ['spam', 'eggs', 'ham']
>>> more = ['fish', 'chips']
>>> food = food + more
>>> food
['spam', 'eggs', 'ham', 'fish', 'chips']

>>> food = ['spam', 'eggs', 'ham']
>>> more = ['fish', 'chips']
>>> food.extend(more)
>>> food
['spam', 'eggs', 'ham', 'fish', 'chips']
```

(the operator makes a new list...)

in

```
>>> food = ['spam', 'eggs', 'ham']
>>> 'eggs' in food
True
>>> 'chicken feet' in food
False
```

reverse()

```
>>> food = ['spam', 'eggs', 'ham']
>>> food.reverse()
>>> food
['ham', 'eggs', 'spam']
```

sort()

```
>>> food = ['spam', 'eggs', 'ham', 'sushi']
>>> food.sort()
>>> food
['eggs', 'ham', 'spam', 'sushi']
```

note:

```
>>> food = ['spam', 'eggs', 'ham', 'sushi']
>>> result = food.sort()
>>> print result
None
```

Sorting

How should this sort?

```
>>> s
[[2, 'a'], [1, 'b'], [1, 'c'], [1, 'a'], [2, 'c']]
```

Sorting

How should this sort?

```
>>> s
[[2, 'a'], [1, 'b'], [1, 'c'], [1, 'a'], [2, 'c']]

>>> s.sort()
>>> s
[[1, 'a'], [1, 'b'], [1, 'c'], [2, 'a'], [2, 'c']]
```

Sorting

You can specify your own compare function:

```
In [279]: s = [[2, 'a'], [1, 'b'], [1, 'c'], [1, 'a'], [2,
In [281]: def comp(s1,s2):
   .....:     if s1[1] > s2[1]: return 1
   .....:     elif s1[1]<s2[1]: return -1
   .....:     else:
   .....:         if s1[0] > s2[0]: return 1
   .....:         elif s1[0] < s2[0]: return -1
   .....:     return 0
In [282]: s.sort(comp)
In [283]: s
Out[283]: [[1, 'a'], [2, 'a'], [1, 'b'], [1, 'c'], [2, 'c']
```

Sorting

Mixed types can be sorted.

"objects of different types always compare unequal, and are ordered consistently but arbitrarily."

`http://docs.python.org/reference/expressions.html#not-in`

Searching

Finding or Counting items

```
In [288]: l = [3,1,7,5,4,3]

In [289]: l.index(5)
Out[289]: 3

In [290]: l.count(3)
Out[290]: 2
```

List Performance

- indexing is fast and constant time: O(1)
- x in s proportional to n: O(n)
- visiting all is proportional to n: O(n)
- operating on the end of list is fast and constant time: O(1)
  append(), pop()
- operating on the front (or middle) of the list depends on n: O(n)
  pop(0), insert(0, v)
  But, reversing is fast. Also, collections.deque

`http://wiki.python.org/moin/TimeComplexity`

Tuples

Tuples are **immutable** sequences

Literal: `(1, 2.0, 'this')`
or
`tuple(something)`

None of the list methods that change the contents

Same indexing, slicing, count, etc.

Lists vs. Tuples

List or Tuples

If it needs to mutable: list

If it needs to be immutable: tuple
(dict key, safety when passing to a function)

Otherwise ... taste and convention

List vs Tuple

Convention:

Lists are Collections (homogeneous):

– contain values of the same type
– simplifies iterating, sorting, etc

tuples are mixed types:

– Group multiple values into one logical thing – Kind of like simple C structs.

List vs Tuple

- Do the same operation to each element?
- Small collection of values which make a single logical item?
- To document that these values won't change?
- Build it iteratively?
- Transform, filter, etc?

List vs Tuple

- Do the same operation to each element? **list**
- Small collection of values which make a single logical item? **tuple**
- To document that these values won't change? **tuple**
- Build it iteratively? **list**
- Transform, filter, etc? **list**

List Docs

The list docs:

`http://docs.python.org/library/stdtypes.html#mutable-sequence-types`

(actually any mutable sequence....)

tuples and commas..

Tuples don't NEED parentheses...

```
In [161]: t = (1,2,3)
In [162]: t
Out[162]: (1, 2, 3)

In [163]: t = 1,2,3
In [164]: t
Out[164]: (1, 2, 3)

In [165]: type(t)
Out[165]: tuple
```

tuples and commas..

Tuples do need commas...

```
In [156]: t = ( 3 )

In [157]: type(t)
Out[157]: int

In [158]: t = (3,)
In [159]: t
Out[159]: (3,)

In [160]: type(t)
Out[160]: tuple
```

LAB

List Lab

`Session2/code/list_lab.rst`

for loops

looping through sequences

```
for x in sequence:
    do_something_with_x
```

for loops

```
In [170]: for x in "a string":
    .....:         print x
    .....:
a

s
t
r
i
n
g
```

range

looping a known number of times..

```
In [171]: for i in range(5):
    .....:     print i
    .....:
0
1
2
3
4
```

(you don't need to do anything with i...

range

`range` defined similarly to indexing

```
In [183]: range(4)
Out[183]: [0, 1, 2, 3]

In [184]: range(2,4)
Out[184]: [2, 3]

In [185]: range(2,10,2)
Out[185]: [2, 4, 6, 8]
```

indexing?

Python only loops through a sequence – not like C, Javascript, etc...

```
for(var i=0; i<arr.length; i++) {
    var value = arr[i];
    alert(i =") "+value);
}
```

indexing?

Use range?

```
In [193]: letters = "Python"

In [194]: for i in range(len(letters)):
   .....:     print letters[i]
   .....:
P
y
t
h
o
n
```

indexing?

More Pythonic – for loops through sequences

```
In [196]: for l in letters:
   .....:     print l
   .....:
P
y
t
h
o
n
```

Never index in normal cases

enumerate

If you need an index – enumerate

```
In [197]: for i, l in enumerate(letters):
   .....:     print i, l
   .....:
0 P
1 y
2 t
3 h
4 o
5 n
```

multiple sequences – zip

If you need to loop though parallel sequences – `zip`

```
In [200]: first_names = ['Fred', 'Mary', 'Jane']

In [201]: last_names = ['Baker', 'Jones', 'Miller']

In [203]: for first, last in zip(first_names, last_names):
   .....:     print first, last
   .....:
Fred Baker
Mary Jones
Jane Miller
```

xrange

`range` creates the whole list

`xrange` is a generator – creates it as it's needed –

a good idea for large numbers

```
In [207]: for i in xrange(3):
   .....:     print i
0
1
2
```

(Python 3 – `range == xrange`)

for

`for` does NOT create a name space:

```
In [172]: x = 10

In [173]: for x in range(3):
    .....:     pass
    .....:

In [174]: x
Out[174]: 2
```

while

`while` is for when you don't know how many loops you need

Continues to execute the body until condition is not `True`

```
while a_condition:
    some_code
    in_the_body
```

while

`while` is more general than `for` – you can always express for as while, but not always vice-versa.

`while` is more error-prone – requires some care to terminate

loop body must make progress, so condition can become `False`

potential error: infinite loops

while vs. for

```
letters = 'Python'
i=0
while i < len(letters):
    print letters[i]
    i += 1
```

vs.

```
letters = 'Python'
for c in letters:
    print c
```

while

Shortcut: recall – 0 or empty sequence is `False`

```
In [13]: l = range(3)
In [14]: while l: # terminates if l is empty
    print l.pop()
2
1
0
```

or

```
while x:     # terminates if x >= 0 on entry
    ...      # do something with x
    x -= 1   # make progress toward 0
```

break

break ends a loop early

```
x = 0
while True:
    print x
    if x > 2:
        break
    x = x + 1
In [216]: run for_while.py
0
1
2
3
```

(This is a pretty common idiom)

break

same way with a `for` loop

```
name = "Chris Barker"
for c in name:
    print c,
    if c == "B":
        break
print "I'm done"
```

```
C h r i s   B
I'm done
```

continue

`continue` skips to the start of the loop again

```
print "continue in a for loop"
name = "Chris Barker"
for c in name:
    if c == "B":
        continue
    print c,
print "\nI'm done"
```

```
continue in a for loop
C h r i s   a r k e r
I'm done
```

continue

`continue` works for a `while` loop too.

```
print "continue in a while loop"
x = 6
while x > 0:
    x = x-1
    if x%2:
        continue
    print x,
print "\nI'm done"
```

```
continue in a while loop
4 2 0
I'm done
```

else again

`else` block run if the loop finished naturally – no `break`

```
print "else in a for loop"
x = 5
for i in range(5):
    print i
    if i == x:
        break
else:
    print "else block run"
```

Strings

A string literal creates a string type

```
"this is a string"
```

Can also use `str()`

```
In [256]: str(34)
Out[256]: '34'
```

or "back ticks"

```
In [258]: `34`
Out[258]: '34'
```

(demo)

The String Type

Lots of nifty methods:

```
s.lower()
s.upper()
    ...
s.capitalize()
s.swapcase()
s.title()
```

http://docs.python.org/library/stdtypes.html#index-23

The String Type

Lots of nifty methods:

```
x in s
s.startswith(x)
s.endswith(x)
...
s.index(x)
s.find(x)
s.rfind(x)
```

http://docs.python.org/library/stdtypes.html#index-23

The String Type

Lots of nifty methods:

```
s.split()
s.join(list)
...
s.splitlines()
```

http://docs.python.org/library/stdtypes.html#index-23

Joining Strings

The Join Method:

```
In [289]: t = ("some", "words","to","join")

In [290]: " ".join(t)
Out[290]: 'some words to join'

In [291]: ",".join(t)
Out[291]: 'some,words,to,join'

In [292]: "".join(t)
Out[292]: 'somewordstojoin'
```

(demo – join)

The string module

Lots of handy constants, etc.

```
string.ascii_letters
string.ascii_lowercase
string.ascii_uppercase
string.letters
string.hexdigits
string.whitespace
string.printable
string.digits
string.punctuation
```

(and the string methods – legacy)

`http://docs.python.org/2/library/string.html#module-string`

String Literals

Common Escape Sequences

```
\\  Backslash (\)
\a  ASCII Bell (BEL)
\b  ASCII Backspace (BS)
\n  ASCII Linefeed (LF)
\r  ASCII Carriage Return (CR)
\t  ASCII Horizontal Tab (TAB)
\ooo  Character with octal value ooo
\xhh  Character with hex value hh
```

(http://docs.python.org/release/2.5.2/ref/strings.html)

Raw Strings

Escape Sequences Ignored

```
In [408]: print "this\nthat"
this
that
In [409]: print r"this\nthat"
this\nthat
```

Gotcha:

```
In [415]: r"\"
SyntaxError: EOL while scanning string literal
```

(handy for regex, windows paths...)

Character Values

Characters in strings are stored as numeric values

"ASCII" values: 1-127

"ANSI" values: 1-255

To get the value:

```
In [109]: for i in 'Chris':
   .....:     print ord(i),
67 104 114 105 115

In [110]: for i in (67,104,114,105,115):
   .....:     print chr(i),
C h r i s
```

(later: unicode!)

Building Strings

Please don't do this:

```
'Hello ' + name + '!'
```

(much)

Building Strings

Do this instead:

```
'Hello %s!' % name
```

much faster and safer:

easier to modify as code gets complicated

`http://docs.python.org/library/stdtypes.html#string-formatting-operations`

String Formatting

The string format operator: %

```
In [261]: "an integer is: %i"%34
Out[261]: 'an integer is: 34'

In [262]: "a floating point is: %f"%34.5
Out[262]: 'a floating point is: 34.500000'

In [263]: "a string is: %s"%"anything"
Out[263]: 'a string is: anything'
```

String Formatting

multiple arguments:

```
In [264]: "the number %s is %i"%('five', 5)
Out[264]: 'the number five is 5'

In [266]: "the first 3 numbers are: %i, %i, %i"%(1,2,3)
Out[266]: 'the first 3 numbers are: 1, 2, 3'
```

String formatting

Gotcha

```
In [127]: "this is a string with %i formatting item"%1
Out[127]: 'this is a string with 1 formatting item'

In [128]: "string with %i formatting %s: "%2, "items"
TypeError: not enough arguments for format string

# Done right:
In [131]: "string with %i formatting %s"%(2, "items")
Out[131]: 'string with 2 formatting items'

In [132]: "string with %i formatting item"%(1,)
Out[132]: 'string with 1 formatting item'
```

String formatting

Named arguments

```
'Hello %(name)s!'%{'name':'Joe'}
'Hello Joe!'

'Hello %(name)s, how are you, %(name)s!' %{'name':'Joe'}
'Hello Joe, how are you, Joe!'
```

That last bit is a dictionary (next week)

String formatting

The format operator works with string variables, too:

```
In [45]: s = "%i / %i = %i"

In [46]: a, b = 12, 3

In [47]: s%(a, b, a/b)
Out[47]: '12 / 3 = 4'
```

So you can dynamically build a format string

Advanced Formatting

The format method

```
In [14]: 'Hello {0} {1}!'.format('Joe', 'Barnes')
Out[14]: 'Hello Joe Barnes!'

In [12]: 'Hello {name}!'.format(name='Joe')
Out[12]: 'Hello Joe!'
```

pick one (probably regular string formatting):
– get comfy with it

LAB

Fun with strings

- Rewrite:
  `the first 3 numbers are: %i, %i, %i"%(1,2,3)`
  for an arbitrary number of numbers...
- write a format string that will take:
  `( 2, 123.4567, 10000)`
  and produce:
  `'file_002 :   123.46, 1e+04'`
- Write a (really simple) mail merge program
- ROT13 – see next slide

`http://docs.python.org/library/stdtypes.html#string-formatting-operations`

LAB

ROT13 encryption

Applying ROT13 to a piece of text merely requires examining its alphabetic characters and replacing each one by the letter 13 places further along in the alphabet, wrapping back to the beginning if necessary

- Implement rot13 decoding
- decode this message:

  Zntargvp sebz bhgfvqr arne pbeare

  (from a geo-caching hint)

Follow Up

Recommended Reading:

- Think Python: Chapt. 9 – 14
- Dive Into Python: Chapt. 6
- String methods: `http://docs.python.org/library/stdtypes.html#string-methods`
- Extra: unicode: `http://www.joelonsoftware.com/articles/Unicode.html`

Do:

- Finish the LABs
- Some CodingBat exercises.
- LPTHW: for extra practice with the concepts – some of: excercises 5 – 14