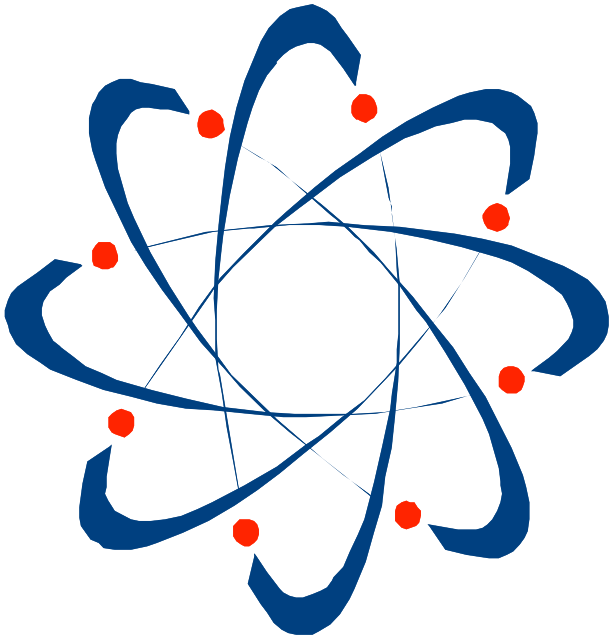
python™ 2 programming

Fundamentals Refresher

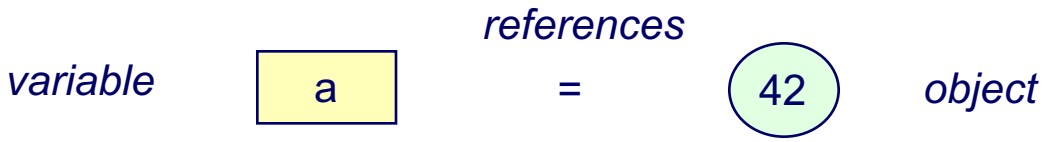
Fundamental Refresher

- **Contents**
  - **Python variables**
    - Type specific methods
  - **Augmented assignments**
  - **Python types**
  - **Numeric types**
  - **Python lists**
  - **Python tuples**
  - **Python dictionaries**



Python variables

- **Python variables are references to objects**



- **Variables are defined automatically**
  - **An undefined variable refers to a special object called None**
- **Variables can be deleted with** `del`
  - An object's memory can be reused when it is no longer referenced
- **Value equals vs Reference equals**
  - == vs is

Python types

- **Everything is an object!**
  - Built in types:
    - Numbers

      `3.142, 42, 0x3f, 0664, 16384L, 0x4E8L`

Sequences

Immutable

    - Strings

      `'Norwegian Blue',"Mr. Khan's bike",r'C:\Numbers'`
    - Tuples

      `(47,'Spam','Major',683,'Ovine Aviation')`

Mutable

    - Lists

      `['Cheddar',['Camembert', 'Brie'],'Stilton']`
    - Dictionaries

      `{'Sword':'Excalibur','Bird':'Unladen Swallow'}`
    - Sets

      `{'Chapman','Cleese','Idle','Jones','Palin'}`
  - Other types include bytes, bytearrays, Boolean, and None

Switching types

- **Sometimes Python switches types automatically**

```
num = 42
pi  = 3.142
num = 42/pi
print num
```

num gets automatic promotion

```
13.367281986
```

- **Sometimes you have to encourage it**
  - **This avoids unexpected changes of type**

```
print "Unused port: "  +  count
TypeError: cannot concatenate 'str' and 'int' objects
```

  - **Use the `str()` function to convert any object to a string**
  - **Use `int()` or `float()` to convert an object to a number**
  - **Other functions available to convert to/from lists and tuples**

```
print "Unused port: " + str(count)
```

Python tuples introduced

- **Tuples are *immutable* (Read-only) objects**
  - **Specified as a comma-separated list of objects, usually inside ( )**
    - The ( ) can sometimes be omitted
    - The comma makes a tuple, not the ( )
  - **Can be indexed in the same way as lists**
    - Starting from 0 on the left or -1 on the right

```
mytuple=('eggs', 'bacon','spam','tea')
print mytuple
print mytuple[1]
print mytuple[-1]
```

```
('eggs', 'bacon', 'spam', 'tea')
bacon
tea
```

  - **Can be reassigned, but not altered**

```
mytuple[2] = 'John'
```

```
TypeError: 'tuple' object does not support item assignment
```

Useful tuple operations

- **Swap references**

```
a,b = b,a
```

- **Set values from a numeric range**

```
Gouda, Edam, Caithness = xrange(3)
```

```
0 1 2
```

- **Repeat values**

```
mytuple = ('a','b','c')
another = mytuple * 4
```

```
('a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c')
```

- **Be careful of single values and the trailing comma**

```
thing = ('Hello')
print type(thing)

thing = ('Hello',)
print type(thing)
```

```
<type 'str'>

<type 'tuple'>
```

Python lists introduced

- **Python lists are similar to arrays in other languages**
  - **Items may be accessed from the left by an index starting at 0**
  - **Items may be accessed from the right by an index starting at -1**
  - **Specified as a comma-separated list of objects inside [ ]**

```
cheese = ['Cheddar', 'Stilton', 'Cornish Yarg']
print cheese[1]
cheese[-1] = 'Red Leicester'
print cheese
```

```
Stilton
['Cheddar', 'Stilton', 'Red Leicester']
```

- **Multi-dimensional lists are just lists containing others**

```
cheese = ['Cheddar', ['Camembert', 'Brie'], 'Stilton']
print cheese[1][0]
```

```
Camembert
```

Tuple and list slicing

- **Slice by start and end *position***
  - **Counting from zero on lhs, from -1 on rhs**

```
mytuple=('eggs', 'bacon','spam','tea','beans')
print  mytuple[2 : 4]
('spam', 'tea')
print mytuple[-4]
bacon
mylist = list(mytuple)
print mylist[1:]
['bacon', 'spam', 'tea', 'beans']
print mylist[:2]
['eggs', 'bacon']
```

start

end+1

  - **List elements may be removed using** `del`


```
cheese = ['Cheddar', 'Camembert', 'Brie', 'Stilton']
del cheese[1:3]
```

```
['Cheddar', 'Stilton']
```

Adding items to a list

- **On the left**

```
cheese[:0] = ['Cheshire', 'Ilchester']
```

- **On the right**

```
cheese += ['Oke', 'Devon Blue']
cheese.extend(['Oke', 'Devon Blue'])
```

Same effect

  - **append can only be used for one item**

```
cheese.append('Oke')
```

- **Anywhere**

```
cheese = ['Cheddar', 'Stilton', 'Cornish Yarg']
cheese.insert(2, 'Cornish Brie')
cheese[2:2] = ['Cornish Brie']
```

Same effect

```
['Cheddar', 'Stilton', 'Cornish Brie', 'Cornish Yarg']
```

Removing items by position

- **Use pop(*index*)**
  - **The index number is optional, default -1 (rightmost item)**
  - **Returns the deleted item**

```
cheese = ['Cheddar', 'Stilton', 'Cornish Yarg']
saved = cheese.pop(1)
print "Saved1:",saved,", Result:",cheese

saved = cheese.pop()
print "Saved2:",saved,", Result:",cheese
```

```
Saved1: Stilton , Result: ['Cheddar', 'Cornish Yarg']

Saved2: Cornish Yarg , Result: ['Cheddar']
```

- **Remember that `del` may also be used**
  - **Does not return the deleted item**
  - **May delete more than one item by using a slice**

Removing list items by content

- **Use the `remove` method**
  - **Removes the leftmost item matching the value**

```
cheese = ['Cheddar', 'Stilton', 'Cornish Yarg',
          'Oke', 'Devon Blue']
cheese.remove('Oke')
```

```
['Cheddar', 'Stilton', 'Cornish Yarg', 'Devon Blue']
```

- **Raises an exception if the item is not found**
  - **Exceptions will be handled later…**

```
cheese.remove('Brie')
```

```
Traceback (most recent call last):
  File "...", line 57, in <module>
    cheese.remove ('Brie')
ValueError: list.remove(x): x not in list
```

Sorting

- `sorted` built-in and `sort` method
  - `sorted` can sort any *sequence*
  - `sorted` returns a sorted list - regardless of the original type
  - `sort` sorts a list in-place
  - Both have the following optional named parameters
    - key=*sort_key* Function which takes a single argument
    - reverse=True Default is False

```
cheese = ['Cornish Yarg', 'Cheddar', 'Stilton']
cheese.sort(key=len)
print cheese
```

```
['Cheddar', 'Stilton', 'Cornish Yarg']
```

```
nums = ['1001','34','3','77','42','9','87']
newnums = sorted(nums, reverse=True)
revnums = sorted(nums, key=int, reverse=True)
```

```
['9', '87', '77', '42', '34', '3', '1001']
['1001', '87', '77', '42', '34', '9', '3']
```

Miscellaneous list methods

- **Count**

list.`count('`*value*`')`	Return the number of occurrences of '*value*'

- **Index**

list.`index('`*value*`')`	Return index position of leftmost '*value*'

- **Reverse**

list.`reverse()`	Reverse a list in place

```
cheese = ['Cheddar', 'Cheshire', 'Stilton', 'Cheshire']
print cheese.count('Cheshire')
print cheese.index('Cheshire')
cheese.reverse()
print cheese
```

```
2
1
['Cheshire', 'Stilton', 'Cheshire', 'Cheddar']
```

List methods

list.append(*item*)	Append *item* to the end of *list*
list.count(*item*)	Return number of occurrences of *item*
list.extend(*items*)	Append *items* to the end of *list* (as +=)
list.index(*item, start, end*)	Return the position of *item* in the *list*
list.insert(*position*, *item*)	Insert *item* at *position* in *list*
list.pop()	Remove and return last item in *list*
list.pop(*position*)	Remove and return item at *position* in *list*
list.remove(*item*)	Remove the first *item* from the *list*
list.reverse()	Reverse the *list* in-place
list.sort(…)	Sort the *list* in-place - arguments are the same as `sorted()`

Sets

- A set is an *unordered* container of object references
  - New with Python 2.4
  - Set items are unique
- Created using the `set` function
  - Add to a set using the `add` method, remove using `remove`

```
setA = set('John')
setB = set('Jane')
print setA, setB

setA.add('y')
setB.remove('e')
print setA, setB
```

```
set(['h', 'J', 'o', 'n']) set(['a', 'J', 'e', 'n'])
set(['y', 'h', 'J', 'o', 'n']) set(['a', 'J', 'n'])
```

  - Here we show a set of characters, but any type may be used

Exploiting sets

- **How do I remove duplicates from a list?**
  - **But we lose the original order**

```
cheese = ['Cheddar', 'Stilton', 'Cornish Yarg',
          'Oke', 'Stilton', 'Cheshire']
cheese = list(set(cheese))
```

`list()` is required, otherwise 'cheese' would now refer to a set

```
['Cornish Yarg', 'Cheshire', 'Cheddar', 'Stilton', 'Oke']
```

- **How do I remove several items from a list?**

```
cheese = ['Cheddar', 'Stilton', 'Cornish Yarg',
          'Oke', 'Stilton', 'Cheshire']
cheese = list(set(cheese) - set(['Stilton', 'Oke']))
```

```
['Cornish Yarg', 'Cheshire', 'Cheddar']
```

Set operators

- Includes set operators
  - Can use a method call instead

```
setA = set('John')
setB = set('Jane')
```

Operator	Method	Returns a new set containing
&	`setA.intersection(setB)`	Each item that is in both sets
|	`setA.union(setB)`	All items in both sets
-	`setA.difference(setB)`	Items in setA not in setB
^	`setA.symmetric_difference (setB)`	Items that occur in one set only

```
print setA & setB
print setA | setB
print setA - setB
print setA ^ setB
```

```
set(['J', 'n'])
set(['a', 'e', 'h', 'J', 'o', 'n'])
set(['h', 'o'])
set(['a', 'e', 'h', 'o'])
```

Python dictionaries introduced

- **A Dictionary object is an unordered collection of objects**
  - **Constructed from { }**

    varname = {*key1*:*object1*,*key2*:*object2*,*key3*:*object3*,…}
  - **Accessed by key**
    - A key is a text string, or anything that yields a text string

    varname[*key*] = *object*

```
mydict = {'Australia':'Canberra', 'Eire'    : 'Dublin',
          'France'   :'Paris',    'Finland':'Helsinki',
          'UK'       :'London',   'US'     :'Washington'}
print mydict['UK']

country = 'Iceland'
mydict[country] = 'Reykjavik'
```

```
London
```

Dictionary values

- **Objects stored can be of any type**
  - **Lists, tuples, other dictionaries, etc…**
  - **Can be accessed using multiple indexes or keys in `[ ]`**

```
mydict = {'UK':['London','Wigan','Macclesfield','Bolton'],
          'US':['Miami','Springfield','New York','Boston']}
print mydict['UK'][2]

homer = 1
print mydict['US'][homer]

mydict['FR'] = ['Paris', 'Lyon', 'Bordeaux', 'Toulouse']
for country in mydict.keys():
    print country, ': ', mydict[country]
```

```
FR :  ['Paris', 'Lyon', 'Bordeaux', 'Toulouse']
US :  ['Miami', 'Springfield', 'New York', 'Boston']
UK :  ['London', 'Wigan', 'Macclesfield', 'Bolton']
```

Removing items from a dictionary

- **To remove a single key/value pair:**
  - `del` *dict*`[`*key*`]`
    - Raises a KeyError exception if the key does not exist
  - *dict*`.pop(`*key*`[,`*default*`])`
    - Returns *default* if the key does not exist

```
>>> fred={}
>>> del fred['dob']
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    del fred['dob']
KeyError: 'dob'
>>> fred.pop('dob',False)
False
```

  - **Also:**
    - *dict*`.popitem()` removes the next key/value pair (used in iteration)
    - *dict*`.clear()` removes all key/value pairs from the dictionary

Dictionary methods

dict.clear()	Remove all items from *dict*
dict.copy()	Return a copy of *dict*
dict.fromkeys(*seq*[,*value*])	Create a new dictionary from *seq*
dict.get(*key*[,*def*ault])	Return the value for *key*, or *default* if it does not exist
dict.has_key(*key*)	True if key exists (*obsolete*)
dict.items()	Return a list of the key-value pairs
dict.keys()	Return a list of the keys
dict.pop(*key*[,*default*])	Remove and return *key*'s value, else return *default*
dict.popitem()	Remove the next item from the dictionary
dict.setdefault(*key*[,*default*])	Add *key* if it does not already exist
dict.update(*dictionary*)	Merge another dictionary into *dict*.
dict.values()	Return a list of the values

Summary

- A Python variable is a reference to an object
- Python variable names are case-sensitive
  - Watch out for leading underscores
- Variables are accessed using operators and methods
  - `dir(`*object*`)` lists the methods available
- Lists are like arrays in other languages
- Tuples are "immutable"
  - But can contain variables
- Dictionaries store objects accessed by key
  - Keys are unique
  - Not ordered

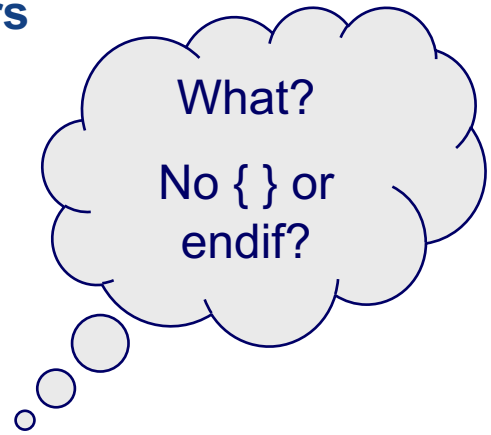
Python conditionals

- **Conditional membership is by *indentation***
  - **Designed for readability**
  - ***Syntax:***

```
if condition:
    statements
elif condition:
    statements
else:
    statements
```

  - **Boolean operators are overloaded by type**
    - **No need for different text or numeric operators**

```
if mylist == mytuple:
    print "Same!"

if 'eggs' in mylist:
    print "Thar be eggs!"
```



What is truth?

- Built-in function `bool()` casts any object to a Boolean
  - False : 0, `None`, empty string, tuple, list, dictionary, set
  - True : everything else
  - Variables True and False are defined
- Use double equal signs (==) to compare values
  - Overloaded for built-in types
  - Use `is` to compare identities of two objects
- Sequence types and dictionaries also support `in`
  - Tests membership of the container

```
lang = ['Perl', 'Python', 'PHP', 'Ruby']
if 'Python' in lang:
    print "Python is there"
```

`in` was introduced at 2.6

Boolean and logical operators

Boolean operators

`<`	**value less than**	*expression* `<` *expression*
`<=`	**value less than or equal**	*expression* `<=` *expression*
`>`	**value greater than**	*expression* `>` *expression*
`>=`	**value greater than or equal**	*expression* `>=` *expression*
`==`	**value equality**	*expression* `==` *expression*
`!=`	**value inequality**	*expression* `!=` *expression*
`is`	**object identity is the same**	*object* `is` *object*

Python 2 also has <> for value inequality

Logical operators

`not`	**logical NOT**	`not` *expression*
`and`	**logical AND**	*expression* `and` *expression*
`or`	**logical OR**	*expression* `or` *expression*

Chained comparisons

- **Useful for a testing a range of values**

```
if 0 < number < 42 < distance:
    print "number and distance are within range"
else:
    print "number and distance are out of range"
```

- **Same as:**

```
if 0 < number and number < 42 and 42 < distance:
    print "number and distance are within range"
else:
    print "number and distance are out of range"
```

- **Can be combined**

```
if 0 < number < 42 and distance != 20:
    ...
```

Sequence and collection tests

- **An empty string, tuple, list, dictionary, set returns False**

```
mylist = [0,1,2,3]
if mylist:
    print "mylist is True"
```

```
mylist is True
```

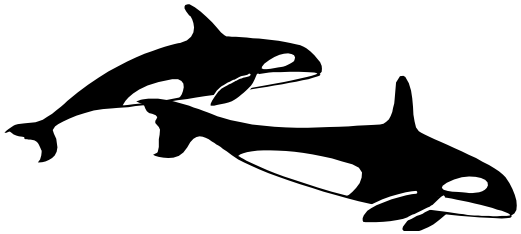
- **Sequences also support built-in `all` and `any`**
  - **`all` returns True if all items in the sequence are true**
  - **`any` returns True if *any* of the items in the sequence are true**

```
mylist = [0,1,2,3]

if not all(mylist):
    print "mylist: not all are True"

if any(mylist):
    print "mylist: at least one item is True"
```

```
mylist: not all are True
mylist: at least one item is True
```

While loops

- **Loop while a condition is true**
  - **Python only supports entry condition loops**
  - **There is no *do...while* loop**



```
while condition:
    loop body
```

- **With all conditionals, membership is by *indentation***

```
line = ""

while line != 'done':
    line = raw_input('Type "done" to complete: ')
    print '<',line,'>'
```

Loop control statements

- **Loop control statements**
  - `continue` **perform next iteration**
  - `break` **exit the loop at once**
  - `pass` **Empty placeholder (no-op)**

- **The else: clause**
  - **Indicates code to be executed when the while condition is false, or when the for list expires**
    - Including when the loop condition is false on entry

```
i = 1; j = 120
while i < 42:
    i = i * 2
    if i > j: break
else:
    print("Loop expired: ",i)
print("Final value: ",i)
```

The `else` clause is not executed if the loop exits using a `break`

```
Loop expired:  64
Final value:  64
```

For loops

- **Iterate through sequence**
  - **Often a list or tuple**
  - **Loop variable holds a copy of each element in turn**

- **As with conditionals, membership is by *indentation***

```
for variable in object:
    loop body
```

```
import sys
for arg in sys.argv :
   print "Cmd line argument:",arg
```

```
C:\Python>for.py Monday Tuesday Wednesday
Cmd line argument: C:\Python\for.py
Cmd line argument: Monday
Cmd line argument: Tuesday
Cmd line argument: Wednesday
```

enumerate

- **Use in loops over any sequence**
  - **Returns a two-item tuple which contains a count and the item at that position in the sequence**

```
for i,arg in enumerate(sys.argv):
    print 'index:',i,'argument:',arg
```

- **… or other object type which supports iteration**
  - **For example, open will open a file and return an iterator**
  - **enumerate also takes an optional *start* parameter**

```
for nr,line in enumerate(open('brian.txt'), start=1) :
    print nr,line,
```

line numbers start from 1, sequences start at 0

```
1 Some things in life are bad
2 They can really make you mad
3 Other things just make you swear and curse.
```

Counting 'for' loops

- **Can use the `xrange()` builtin**

  `xrange`([*start*], *stop*[, *step*])

```
for i in xrange(0,len(some_list)):
    if some_list[i] > 42: some_list[i] += 1
```

  - **But this maintains its own iterator**

```
for i in xrange(0,len(some_list)):
    print some_list[i]
```

  - **Use a system generated one instead**

```
for num in some_list:
    print num
```

- **But an index is needed to alter the sequence…**

```
for i,num in enumerate(some_list):
    if num > 42: some_list[i] += 1
```

Zipping through multiple lists

- **The zip builtin returns a list of tuples**
  - **Can consume a lot of memory**
  - **Useful for stepping through parallel lists**

```
farms   = ['Home Farm', 'Muckworthy',
           'Scales End', 'Brown Rig']
squirls = [42, 12, 2, 0]
rabbits = [395, 68, 57, 32]
moles   = [12, 8, 0, 29]

for (f, s, r, m) in zip(farms, squirls, rabbits, moles):
    print 'Total for',f,':',s + r + m
```

```
Total for Home Farm : 449
Total for Muckworthy : 88
Total for Scales End : 59
Total for Brown Rig : 61
```

*A squirl is a truncated squirrel

Conditional expressions

- **Shorthand for conditionals**
  - **Added at Python 2.5**

expr1 **`if`** *boolean* **`else`** *expr2*

```
if i > j:
    print "i gt j"
else:
    print "i lt j"
```

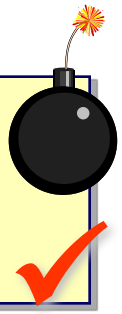
```
i = 42
j = 3

print "i gt j" if i > j else "i lt j"
```

- **No : and elif not allowed**

```
-1 if a < b else (+1 if a > b else 0)
```

- **Beware of precedence**

```
a = 54
answer = a + 5 if a < 42 else 0

answer = a + (5 if a < 42 else 0)
```



Summary

- **Python has the usual Boolean and logical operators**
  - **Be careful of types**

- **Basic flow control statements :**

  if *condition*:

  indented statements

  while *condition*:

  indented statements

  for *target* **in** *object*:

  indented statements

- **Terminate a process using** `exit`

Functions

```
def fn(x,y):
        return x+y
res=fn(10,20)
```

```
def add100(x,y=0,z=0):
        return 100 + x + y + z
```

Python Strings

- **Python does not do interpolation**
  - **Single and double quotes have the same effect**

```
print 'hello\nworld'
```

⇔

```
print "hello\nworld"
```

  - **Use " when you have embedded ', and vice versa**

- **With embedded quotes or new-lines, use triple quotes**

```
>>> html = """
<tr>
      <td><font color="#690000"><b>Username :</b></font></td>
      <td><input type='textbox' name='username'></td>
</tr>
"""
```

```
'\n<tr>\n\t<td><font color="#690000"><b>Username :</b></font></td>\n
\t<td><input type=\'textbox\' name=\'username\'></td>\n</tr>\n'
```

wrapped around

String methods

- **The `string` module is now mostly replaced by methods**
- **Some useful string functions and methods**

String to a number	`int`	`int("42")`
Object to a string	`str`	`str(42)`
Object to a string	`repr`	`repr(obj)` - see notes
Number of characters	`len`	`len(name)`
Convert to lower case	`lower`	*`str`*`.lower()`
Replace a sub-string	`replace`	*`str`*`.replace('old','new')`
Remove trailing chars	`rstrip`	*`str`*`.rstrip()`
Search for a sub-string (returns the offset)	`find`	*`str`*`.find('cheese')`

- **Overloaded * operator**

```
>>> 'Spam ' * 4
```

Mandatory Monty Python reference

```
'Spam Spam Spam Spam '
```

String tests

- **Remember the `in` operator**

```
if substr in string:
```

- **Testing a string type can often be done with a method**
  - **Regular Expressions can also be used, but can be slow**

count
endswith
isalnum
isalpha
isdigit
islower
isspace
istitle
isupper
startswith

```
txt = 'hello world'
print txt.count('o')
if txt.startswith('hell'):
    print "It's hell in there"
if txt.isalpha():
    print 'string is all alpha'
txt = ' \t\r\n'
if txt.isspace():
    print 'string is whitespace'
```

```
2
It's hell in there
string is whitespace
```

String formatting

- **The % operator is overloaded for strings**
  - **Like `sprintf` in some other languages**

format_string **% (***argument_list***)**

  - ***format_string***
    - contains text and format specifiers, prefixed %
    - describe format of the plugged-in value
  - ***argument_list***
    - contains text or variables to be plugged-in

- **Format specifiers**

%s	string	%o	octal
%c	character	%x	lowercase hex
%d	decimal	%X	uppercase hex
%i	integer	%%	literal %
%u	unsigned int		

%e, %E, %f, %g, %G - alternative floating point formats

String formatting example

- **Common conversion specifiers:**
  - `%d` Treats the argument as an integer number
  - `%s` Treats the argument as a string
  - `%f` Treats the argument as a float (and rounds)

```
planets = {'Mercury' : 57.91,
           'Venus'   : 108.2,
           'Earth'   : 149.597870,
           'Mars'    : 227.94}

for i,key in enumerate(planets.keys(),1):
    print "%2d %-10s %06.2f Gm" % (i,key,planets[key])
```

```
 1 Earth      149.60 Gm
 2 Mercury    057.91 Gm
 3 Mars       227.94 Gm
 4 Venus      108.20 Gm
```

Other string formatting aids

- **Often more efficient, very often easier**
  - *string*`.capitalize()`
  - *string*`.lower()`/*string*`.upper()`
  - *string*`.center()`
  - *string*`.ljust()`
  - *string*`.rjust()`
  - *string*`.zfill()`

```
str = 'hello'

print str.capitalize()
print str.upper()
print '<'+str.center(12)+'>'
print '<'+str.ljust(12)+'>'
print '<'+str.rjust(12)+'>'
print '<'+str.zfill(12)+'>'
```

```
Hello
HELLO
<   hello    >
<hello       >
<       hello>
<0000000hello>
```

Slicing a string

- **A Python string is an immutable sequence type**
  - **Slicing is the same for all sequence types**

- **Slice by start and end *position***
  - **Counting from zero on lhs, from -1 on rhs**

```
#        0123456789012345678901234567890123 4
text = "Remarkable bird, the Norwegian Blue"
print text[11:14]
bir
print text[-7:-1]
an Blu
```

end+1

start

  - **Start and end positions may be defaulted**

```
print(text[ : 14])
Remarkable bir
print(text[-7: ])
an Blue
```

String methods - split and join

- **String to a list - split**
  - *string*`.split([` *separator*`[,` *max_splits*`]])`
    - If *separator* is omitted, split on one or more white-space
    - If *max_splits* is omitted, split the whole string
    - *string*`.splitlines()` is useful on lines from files

- **Sequence to a string - join**
  - *separator*`.join(`*sequence*`)`

```
line = 'root::0:0:superuser:/root:/bin/sh'
elems = line.split(':')

elems[0] = 'avatar'
elems[4] = 'The super-user (zero)'
line = ':'.join(elems)
print line
```

```
avatar::0:0:The super-user (zero):/root:/bin/sh
```

Python operators

`or`	logical OR
`and`	logical AND
`not`	logical NOT
`< <= > >=`	comparison operators
`== !=`	equality operators
`is`	object identity test
`in`	object membership test
`|  ^`	binary OR, XOR
`&`	binary AND
`<<  >>`	binary shift
`- +`	subtract, add
`*  / // %`	multiply, divide, integer-divide, modulo
`~ **`	complement, exponentiation

Python reserved words

- **The following are illegal as variable or function names in Python**

and	as*	assert	break	class	continue
def	del	elif	else	except	exec~
finally	for	from	global	if	import
in	is	lambda	not	or	pass
print^	raise	return	try	while	with*
yield					

* version 2.6 and later

~ not in version 3.0

Summary

- Python variables are not embedded inside quotes
  - But characters like \r\n\t can be
  - No difference between ' and "
  - Use three quotes for multi-line text
- Several methods available on a string
  - Many for conversions
- Formatting uses the % operator
- Strings can be sliced[start:end+1]
  - As can other sequences
- Split a string with split, join items in a list with join