

Strings

```
>>> a = "going to class"  
>>> b = 'going to class'  
>>> c = """going to class"""  
>>> d = r'going to class'
```

'\n' is a carriage

return

raw format

using **Triple quotes**

```
In [39]: a="""going  
.....:         to school  
.....:         late  
.....:         """
```

```
In [40]: a  
Out[40]: 'going \n         to school\n         late\n'
```

```
In [41]: print a  
going  
         to school  
         late
```

Advantage of raw format

- I want to encode the string: “path\to\file”
- “\” is a special character and one must do:“\\”

```
In [72]: a='\a\b\c'
```

```
In [73]: a
```

```
Out [73]: '\x07\x08\\c'
```

```
In [74]: print a
\c
```

```
In [75]: a=""'\a\b\c'""
```

```
In [76]: print a
\c
```

```
In [77]: a=r'\a\b\c'
```

```
In [78]: print a
\a\b\c
```

```
In [79]: a='\\a\\b\\c'
```

```
In [80]: print a
\a\b\c
```

Booleans

- A boolean is either **True** or **False**
- In some languages, -1 if true, all else is false, in others, 0 is false, all else is true
- In Python : the number 0 is false, all other numbers are true. **Do not assume this!!!**
- **None, (), [], 0, "" returns false**
 - `bool(None) ==> False`
 - `bool([]) ==> False`
 - `bool(34) ==> True`
 - `bool(None or 34) ==> True`
 - `bool(34 and (not 0 or "")) ==> True`

What is None?

```
In [1]: bool(trip)
```

```
NameError  
call last)
```

Traceback (most recent

```
<ipython-input-1-2b29e319ed42> in <module>()  
----> 1 bool(trip)
```

```
NameError: name 'trip' is not defined
```

```
In [2]: trip=None  
In [3]: bool(trip)  
Out [3]: False
```

```
In [4]: trip=""  
In [5]: bool(trip)  
Out [5]: False
```

```
In [6]: trip=3  
In [7]: bool(trip)  
Out [7]: True
```

- **None** is the absence of definition
- **""** is the empty string

Lists

- A collection of objects
- There is an order
 - `a[0]` comes before `a[1]`
- List elements can be modified (**mutable**)
- Heterogeneous (strings, ints, floats, functions)

Lists

- `a = []` #empty list
- `a.extend([3,4])` # a = [3,4]
- `a.append([3,4])` # a = [3,4,[3,4]] (add single element)
- `a.extend([3,4])` # a = [3,4,[3,4],3,4] (add elements)
- `a[1] = 'pyth'` # a = [3,'pyth',[3,4]]
- mutable
- heterogeneous
- `type(a)` # <type 'list'>

List Initialization

- Use the “*” operator
 - `a = [3] * 10 ==> [3,3,....,3]`
 - `b = [3,4,5] * 7 ==> [3,4,5,3,4,5,....,3,4,5]`
 - `c = 'hu' * 5 ==> 'huhuhuhu'`
- Use an iterator
 - `a = list(xrange(5)) # [0,1,2,3,4]`
 - `print xrange(5) # xrange(5)`
 - `type(xrange(5)) # <type 'xrange'> (iterator)`

```
class xrange(object)
| xrange(stop) -> xrange object
| xrange(start, stop[, step]) -> xrange object
|
| Like range(), but instead of returning a list, returns an object that
| generates the numbers in the range on demand. For looping, this is
| slightly faster than range() and more memory efficient.
```


Special lists

- `range(5)` # returns 0,1,2,3,4
- `xrange(5)` # iterator object
- `dir([])`
['**__add__**', '__class__', '__contains__', '__delattr__', '__delitem__', '__delslice__', '__doc__', '__eq__', '__format__', '__ge__', '__getattr__', '__getitem__', '**__getslice__**', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '**__mul__**', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__setslice__', '__sizeof__', '__str__', '__subclasshook__', '**append**', '**count**', '**extend**', 'index', '**insert**', 'pop', 'remove', '**reverse**', '**sort**']
- `a = range(5)`
`a.reverse().sort()` # in place reversion followed by a sort

References

```
>>> a = range(8)
>>> a
[0, 1, 2, 3, 4, 5, 6, 7]
>>> b = a
>>> b[5] = 'class'
>>> a
[0, 1, 2, 3, 4, 'class', 6, 7]
>>>
```

b is a reference to **a**

any change to an element to **b** also changes **a**

Splicing

```
>>> a = range(8)
>>> a
[0, 1, 2, 3, 4, 5, 6, 7]
>>> b = a[:]
>>> b[5] = 'class'
>>> a
[0, 1, 2, 3, 4, 5, 6, 7]
>>> b
[0, 1, 2, 3, 4, 'class', 6, 7]
```

b is a copy of **a**
Changing an element of **b**
does not change **a**

`a[:]` is an example of a splice
Splices are **copies** of a *subset* of the original array

Splicing

- `a = range(5)`
- `a[3:5]`
- `a[3:]`
- `a[-3]`
- `a[-1]` # last element
- `a[-3:-1]` #

List Errors

- `c[3] = 4`
 - `# Name error: c not defined`
- `c = []`
- `c[3] = 2`
 - `# Index error: list assignment index out of range`

Sequences, Tuple

- A sequence is similar to a list, *except* that it cannot be modified
 - immutable

Sequence

- Immutable
- Cannot be changed
- $a = (2,3,5)$
- $a[1] = 3$ # exception
- $a = (3)$ # not a sequence
- $a = 3,$ # or $(3,)$ is a sequence

Sequence

Immutable object

```
>>> a = (1,2,3)
>>> type(a)
<type 'tuple'>
>>> dir(a)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattr__', '__getitem__',
'__getnewargs__', '__getslice__', '__gt__', '__hash__', '__init__',
'__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmul__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'count',
'index']
```



```
>>> a = ('notes',3,-34,7)
>>> a[1] = 3          # immutable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> a + (5,6,7)      # a.__add__((5,6,7))
('notes', 3, -34, 7, 5, 6, 7)
>>> len(a)          # a.__len__
4
>>> a*3             # a.__mul__(3)
('notes', 3, -34, 7, 'notes', 3, -34, 7, 'notes', 3, -34, 7)
>>> a[5]           # a.__getitem__(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
>>> a[2]
-34
>>> a
('notes', 3, -34, 7)
```

Dictionaries

- A list can contain anything, but there is an order: the list can be indexed.
- A dictionary (also called *hash*) is a collection of (key:value) pairs
- There is no indexing
- The key can be any immutable object
 - int, float, long, sequence, string
- The value can be any object (mutable, immutable)
 - list, class, function, etc. s

Dictionary

- `a = {}` # or `a = dict()` (not common)
- `c = {'1': 'gordon', 2: 'fran'}`
 - `c['1']` # 'gordon'
 - `c[1]` # error (key not defined)
- `a[3] = 'gordon'`
- `a['egg'] = 'steamed'`
- `a[(3,4,5)] = ['class', [3,4,5], 6]`
- `b = a[(3,4,5)][2]` returns 6
- `dict[key] = value`

Dictionary

- `variable[key] = value`
- key can be:
 - any immutable object
 - string, int, float, sequence

Dictionary

- `dir({})`
- `['__class__', '__cmp__', '__contains__', '__delattr__', '__delitem__', '__doc__', '__eq__', '__format__', '__ge__', '__getattr__', '__getitem__', '__gt__', '__hash__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'clear', 'copy', 'fromkeys', 'get', 'has_key', 'items', 'iteritems', 'iterkeys', 'itervalues', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']`

```
>>> a = {}
>>> a[3] = 'gor'
>>> a['frank'] = 'code'
>>> a['grow'] = 35.5
>>> a.keys()
['frank', 3, 'grow']
>>> del a['grow']
>>> a.keys()
['frank', 3]
>>> len(a)
2
>>> c = a['grow']
>>> c = a.__getitem__('frank')
>>> c
'code'
```

Dictionary Members

```
>>> a = {}
>>> a[3] = (3,5,6)
>>> a[('joe', 34)] = 'exam'
>>> a['area'] = 3.56
>>> a.keys()
[3, ('joe', 34), 'area']
>>> a.values()
[(3, 5, 6), 'exam', 3.5600000000000000]
>>> a.itervalues()
<dictionary-valueiterator object at 0x374a80>
>>> a.has_key((3,5))
False
>>> a.has_key((3,5,6))
False
>>> a.has_key(('joe',34))
True
>>>
```

```
>>> a={}
>>> a[1]='peter'
>>> a[2]='jasmin'

>>> a.items()
[(1, 'peter'), (2, 'jasmin')]

>>> a.keys()
[1, 2]
>>> a.values()
['peter', 'jasmin']
```

Sets

- A set is a collection of objects
- There is no order to these objects
- Each element in a set is unique
 - contrary to a list
 - `a = [1,2,3,3,3]` contains the integer 3 three times
 - `s = set((1,2,3,3,3))` or `set([1,2,3,3,3])` returns `set([1,2,3])` (the other two 3's are removed)

Set

- `a = set()`
- `dir(a)`
- `['__and__', '__class__', '__cmp__', '__contains__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__', '__getattr__', '__gt__', '__hash__', '__iand__', '__init__', '__ior__', '__isub__', '__iter__', '__ixor__', '__le__', '__len__', '__lt__', '__ne__', '__new__', '__or__', '__rand__', '__reduce__', '__reduce_ex__', '__repr__', '__ror__', '__rsub__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__xor__', 'add', 'clear', 'copy', 'difference', 'difference_update', 'discard', 'intersection', 'intersection_update', 'isdisjoint', 'issubset', 'issuperset', 'pop', 'remove', 'symmetric_difference',`

Unique words with Sets

- Assume the following task:
 - given two books, what are the words common to both (plurals and other inflections count as separate words)
- Solution
 - collect all the words from book *A* into setA, and collect all the words from book *B* into setB. The required set of unique words is then simply
 - `unique_words = setA.intersection(setB)`

Loops

```
>>> a = [10,'gor',-35,3132]
>>> for i in a:      # list is explicit
...     print i
...
10
gor
-35
3132

>>> for i in range(3): # explicit list
...     print i
0
1
2
range(3)      # explicit list
[0, 1, 2]
```

Loops

```
>>> xrange(3) # iterator (list is not explicit)
xrange(3)
>>> for i in xrange(3):
...     print i
...
0
1
2
```

Iterators

- Proxy for a list
- The list is implicit
- One can construct our own iterators using *yield*
- Iterators are used in for loop constructs
- Useful when lists are huge (uses up memory) and the elements should be accessed one at a time

Yield

- Wait until we discuss functions
- But here is a quick example for illustration

```
def even_items(n):  
    for item in xrange(n):  
        yield 2*item+1  
  
for i in even_items(12):  
    print i
```

prints out all odd numbers less than 24

Next Lecture

- Functions