Truth values

We already saw that empty means FALSE in python.
The same applies to zero too.

```
1  object = 'somestring'
2
3  if len(object) > 0:
4      print ('my_object is not empty')
5
6  if len(object):
7      print ('my_object is not empty')
8
9  if object != '':
10     print ('my_object is not empty')
11
12 if object:
13     print ('my_object is not empty')
```

Functions

We have already seen functions. But only the simplest forms. We can have functions

- With arguments having default values
- With keywords as arguments
- With multiple arguments.

```
>>> def myfoo(bar, foobar=True):
...     print(bar)
...     if foobar:
...         print("ha ha ha!")
...
>>> myfoo("hello")
hello
ha ha ha!
>>> myfoo("hello", foobar=False)
hello
>>>
```

Default values taken only once

- The default value of the parameter is initialised only once and it stays the same if not specifically called. Look at the following example.

```
>>> def add(this, tothat=()):
...     for e in this:
...         tothat.append(e+1)
...     return tothat
...
>>> add((23, 34))
(24, 35)
>>> add((23, 34))
(24, 35, 24, 35)
>>> add((23, 34))
(24, 35, 24, 35, 24, 35)
>>> add((23, 34))
(24, 35, 24, 35, 24, 35, 24, 35)
>>> add((23, 34), (1, 2))
(1, 2, 24, 35)
>>>
```

Multiple Arguments

- Functions with a *-ed argument can have multiple arguemnt.
- The arguments would be packed in a tuple
- The *-ed argument must follow the other typed of arguments.

```
>>> def mularg(i, j, *rest):
...     print(i+j)
...     for k in rest:
...         print(k)
...
>>> mularg(1, 2)
3
>>> mularg(1, 2, 4)
3
4
>>> mularg('hello', 'world',
           'this', 'is', 'cool!')
helloworld
this
is
cool!
>>>
```

Docstrings

- Strings surrounded by three quotes at the beginning of functions could be used for documentation purposes.
- These strings contain newlines in them.

```
>>> def simpledoc():
...     """This is a simple hello
...        world program - just to reveal
...        the beauty of docstrings"""
...     print("Hello World")
...
>>> simpledoc.__doc__
'This is a simple hello\n        world program - j
the beauty of docstrings'
>>> print simpledoc.__doc__
This is a simple hello
       world program - just to reveal
       the beauty of docstrings
>>> help(simpledoc)
...
```

With expression

- Files are to be always closed after use.
- A keyword named `with`
- Using `with` helps automatic closing of files after use.
- The object which is used with `with` must have the methods - `__enter__` and `__exit__` implemented

```
1
2
3
4 with open(filename) as f:
5     for line in f:
6         print line
```

Flatten A List

```
1  def flatten(e):
2      if not e:
3          return ()
4      if not isinstance(e, list):
5          return (e)
6      return flatten(e[0]) + flatten(e[1:])
7  ------------------------------------------------------------
8  def flatten(vl):
9      fl = ()
10     for el in vl:
11         if isinstance(el, list):
12             for e in flatten(el):
13                 fl += (e)
14         else: fl += (el)
15     return fl
```

String Theory

- The strings in python contains many methods. One of them is `find` which returns the position of a substring
- But if we need only to check if the substring is present in a big string, we don't need to use that. (More readable code)
- `split` and `join`: These are two string methods which are very useful.

```
>>> string = 'Hi there'
>>> if string.find('Hi') != -1:
...     print('Success!')
...
Success!
>>> if 'Hi' in string:
...     print ('Success!')
...
Success!
>>>
>>> mystr = 'this is a one two three string'
>>> words = mystr.split()
>>> words
('this', 'is', 'a', 'one', 'two', 'three', 'strin
>>> '*'.join(words)
'this*is*a*one*two*three*string'
>>>
```

Filter, Map and Reduce

`func_tool(function, sequence)`

- `filter`: Filter accepts two parameters, one is a function and the second one a sequence. It returns a list of the elements of the sequence for which the function is TRUE.
- `map`: The returned list would be the results of applying the function to each member of the sequence.
- `reduce`: Initially, the function is applied to the first two elements of the sequence, and the result used as the parameter along with the next elements of the sequence.

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> list(filter(f, range(2, 25)))
(5, 7, 11, 13, 17, 19, 23)
>>> def cube(x): return x*x*x
...
>>> list(map(cube, range(1, 11)))
(1, 8, 27, 64, 125, 216, 343, 512, 729, 1000)
>>> seq = range(8)
>>> def add(x, y): return x+y
...
>>> list(map(add, seq, seq))
(0, 2, 4, 6, 8, 10, 12, 14)
```

```
>>> def add(x,y): return x+y
...
>>> from functools import reduce
>>> reduce(add, range(1, 11))
55
>>> def sum(seq):
...     def add(x,y): return x+y
...     return reduce(add, seq, 0)
...
>>> sum(range(1, 11))
55
>>> sum([])
0
```

ZIP

In case we need to combine two lists, How do we do it?
How do we create a dictionary from two lists?

```
>>> l = (x for x in range(1, 10))
>>> k = (y for y in range(90, 99))
>>> l
(1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> k
(90, 91, 92, 93, 94, 95, 96, 97, 98)
>>>
>>>
>>> lk = ((l(x), k(x)) for x in range(len(l)))
>>> lk
((1, 90), (2, 91), (3, 92), (4, 93), (5, 94),
 (6, 95), (7, 96), (8, 97), (9, 98))
```

```
>>>
>>> lkl = list(zip(l, k))
>>> lkl
((1, 90), (2, 91), (3, 92), (4, 93), (5, 94),
          (6, 95), (7, 96), (8, 97), (9, 98))
>>>
>>> lkd = dict(lkl)
>>> lkd
{1: 90, 2: 91, 3: 92, 4: 93, 5: 94,
        6: 95, 7: 96, 8: 97, 9: 98}
>>>
```

```
>>> t = (5,6,7)
>>> d = dict(list(zip(t, range(len(t)))))
>>> d
{5: 0, 6: 1, 7: 2}

>>> d = dict(enumerate(t))
>>> d
{0: 5, 1: 6, 2: 7}

>>> d = dict((y,x) for x,y in enumerate(t))
```

How does the knight jump 64 in 64

Regular Expressions Basics

- Alphabets
- Operators : $*, +, ?, |$
- Examples : $(0|1)*, a(bc|d)*, a+$

In Python

In python, there exists a module for regular expressions. Here we can see some example symbols

- . - Stands for any character
- \w - matches all alphanumeric characters and '_'
- \W - matches anything which is not \w
- \d - matches digits

Using them

The standard way to use regular expressions in python is as follows.

- Compile the expression to a pattern object.
- Then the object is matched against the test.
- If successfully matches, a Match object is returned, with the relavant information.

```
>>> import re
>>> pattern = re.compile('a(a*)b')
>>> text = 'xyzaaaab3sf'
>>> matcher = pattern.search(text)
>>> print(matcher.group())
aaaab
>>>
>>>
```

More in next lecture

Powerful tools

We saw lambda functions and the other functional programming methods of python. They can be used together to have very powerful routines with simple code.

```
1 def add(a,b): return a+b
2
3 add2 = lambda a,b: a+b
4
5 squares = list(map(lambda a: a*a, (1,2,3,4,5)))
6
```

Syntax

A lambda function has the syntax: `lambda variable(s) : expression`

variable(s)	a comma-separated list. No keywords; No parentheses.
expression	python expression. Scope: local scope and variable(s). This is what the function returns.

```
>>> foo = (2, 18, 9, 22, 17, 24, 8, 12, 27)
>>>
>>> print(list(filter(lambda x: x % 3 == 0, foo))
(18, 9, 24, 12, 27)
>>>
>>> print(list(map(lambda x: x * 2 + 10, foo)))
(14, 46, 28, 54, 44, 58, 26, 34, 64)
>>>
>>> print(reduce(lambda x, y: x + y, foo))
139

>>> nums = range(2, 50)
>>> for i in range(2, 8):
...     nums = list(filter(lambda x:
                    x == i or x % i, nums))
...
>>> print nums
```

```
18 (2, 3, 5, 7, 11, 13, 17, 19,
19          23, 29, 31, 37, 41, 43, 47)
```

Iterators

Iterators

For loop is used a lot in Python. One can iterate over almost every type of collection.
How is this made possible?

```
1  for element in (1, 2, 3):
2      print element
3  for element in (1, 2, 3):
4      print element
5  for key in {'one':1, 'two':2}:
6      print key
7  for char in "123":
8      print char
9  for line in open("myfile.txt"):
10     print line
```

Iterators

- When for statement is called, a method `iter` is called on the object.
- This returns an object on which, the method `next` is implemented (which can go through the items)
- `next` keeps on giving elements, one by one.
- When there are no more elements, an exception `StopIteration` is raised. (Loop stops)

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> it.next()
'a'
>>> it.next()
'b'
>>> it.next()
'c'
>>> it.next()

Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    it.next()
StopIteration
```

How to make Iterable Classes?

To make a collection (personal class) iterable:

- It needs to have the method `__iter__` implemented. This is the function which enables `iter` to be called.
- `__iter__` should return and object with `next` implemented.
- Example below.

```
1
2  class Reverse:
3      """Iterator for looping over a
4       sequence backwards"""
5      def __init__(self, data):
6          self.data = data
7          self.index = len(data)
8      def __iter__(self):
9          return self
10     def next(self):
11         if self.index == 0:
12             raise StopIteration
13         self.index = self.index - 1
14         return self.data[self.index]
```

```
>>> for char in Reverse('spam'):
...     print(char)
...
m
a
p
s
```

Advantages/Disadvantages

- When we have Iterator implemented on an object, the for loop would not copy the object. So, especially for large collections, this is advantageous.
- Troubles: When the list (collection) has to be changed, an iterator can lead to catastrophe.
- In case of lists, use slicing. (Example)

I want all the squares upto 121 (not single digit) and I want also every double digit square + 30.

```
>>> lis = (x**2 for x in range(4, 12))
>>>
>>>
>>> for i in lis:
...     if i < 100:
...         lis.append(i+30)
...
>>> lis
(16, 25, 36, 49, 64, 81, 100, 121, 46,
 55, 66, 79, 94, 111, 76, 85, 96, 109,
 124, 106, 115, 126)
>>>
```

```
>>>
>>> lis = (x**2 for x in range(4, 12))
>>> for i in lis(:):
...     if i < 100:
...         lis.append(i+30)
...
>>> lis
(16, 25, 36, 49, 64, 81, 100, 121, 46,
 55, 66, 79, 94, 111)
>>>
```

Applied/Used

- In for slices
- In list comprehensions, for expressions
- in If operators `if x in this`
- In almost all the collections.
- More efficient than copying.

In Dicts

- `iter(d)` gives an iterator over the keys of the dict
  1. `d.iterkeys`
  2. `d.itervalues`
  3. `d.iteritems`
- Iterators over `d.keys, d.values, d.items`
- No lists are created.

Pickle

- Module in python
- Serialisation and de-serialisation of python objects
- Serialisation : converting to a byte stream.
- The reverse to get the object back.

Pickling

- Marshalling [1]
- Serialisation
- Flattening
- Pickling / Unpickling

[1] Nothing to do with the object Marshal

cPickle and Marshal

- cPickle is the very same module implemented in C
- cPickle, yes, it is fast: about 1000 times.
- Pickle keeps track of serialisation and there is no repeated serialisation (unlike marshal)
- Shelve (for dictionaries)

How to Pickle?

- `pickle.dump(obj, file)`
- `pickle.load(file)`
- `pickle.dumps(obj)`
- `pickle.loads(str)`

A write permission to the file is required for the `dump` to work. Also, the file should have `read` and `readline` functions implemented for the `load` to be functional.

What All?

- None, True, and False
- integers, long integers, floating point numbers, complex numbers
- normal and Unicode strings
- Collections with only picklable objects
- functions defined at the top level of a module
- built-in functions defined at the top level of a module
- classes that are defined at the top level of a module

```
>>> import pickle
>>> class Foo:
...     attr = 'a class attr'
...
>>> picklestring = pickle.dumps(Foo)
>>>
>>> x = Foo()
>>> picklestring2 = pickle.dumps(x)
>>>
>>> picklestring
'c__main__\nFoo\np0\n.'
>>> picklestring2
'(i__main__\nFoo\np0\n(dp1\nb.'
>>>
>>> y = pickle.loads(picklestring2)
>>>
>>> isinstance(y, Foo)
```

```
18 True
19 >>> isinstance(x, Foo)
20 True
21 >>>
```

Shelves

- A shelve is a persistent dictionary object in python
- A dictionary in the secondary storage
- Could be opened and used as needed.
- `open` and `close` are the usual methods needed.

```
>>> import shelve
>>> d = shelve.open("myfile.shelf")
>>> d('lala') = 'booboo'
>>> d('kiki') = 'myamya'
>>> d
{'lala': 'booboo', 'kiki': 'myamya'}
>>> d('xx') = range(4)
>>> d
{'lala': 'booboo', 'xx': (0, 1, 2, 3),
                   'kiki': 'myamya'}
>>> d.close()
>>>
(sadanand@lxmayr10
myfile.shelf.bak   myfile.shelf.dat
                   myfile.shelf.dir
(sadanand@lxmayr10
>>> import shelve
```

```
>>> d = shelve.open("myfile.shelf")
>>> d
{'kiki': 'myamya', 'xx': (0, 1, 2, 3),
                  'lala': 'booboo'}
```

Kiki Booba

Kiki

Booba

Problems

- BST
- Eval Expression
- Set Operations with lambda/map/filter
- Methods for Order Statistics
- Reversal
  - 3 for reversal (string/list)
  - Use that to implement `rev` functionality of UNIX