Introduction to Programming in Python (2)

Steve Renals
s.renals@ed.ac.uk

ICL — 28 September 2005

Dictionaries

Dictionaries are

- Addressed by *key*, not by offset
- *Unordered* collections of arbitrary objects
- Variable length, heterogenous (can contain any type of object), nestable
- *Mutable* (can change the elements, unlike strings)

Dictionaries

Dictionaries are

- Addressed by *key*, not by offset
- *Unordered* collections of arbitrary objects
- Variable length, heterogenous (can contain any type of object), nestable
- *Mutable* (can change the elements, unlike strings)
- Think of dictionaries as a set of key:value pairs
- Use a key to access its value

(*Learning Python*, chapter 7)

Dictionary example

```
level = {'icl' : 9, 'pmr' : 11, 'inf2b' : 8}
x = level['pmr'] # 11
n = len(level) # 3

flag = level.has_key('inf2b') # True
l = level.keys() # ['inf2b', 'pmr', 'icl']

level['dil'] = 11 # {'dil': 11, 'inf2b': 8, 'pmr': 11, 'icl': 9}
level['icl'] = 10 # {'dil': 11, 'inf2b': 8, 'pmr': 11, 'icl': 10}

l = level.items() # [('dil', 11), ('inf2b', 8), ('pmr', 11), ('icl',
10)]
l = level.values() # [11, 11, 8, 10]
```

Notes on dictionaries

- Sequence operations don't work: dictionaries are *mappings*, not sequences
- Dictionaries have a *set* of keys: only one value per key
- Assigning to a new key adds an entry
- Keys can be any immutable object, not just strings

Notes on dictionaries

- Sequence operations don't work: dictionaries are *mappings*, not sequences
- Dictionaries have a *set* of keys: only one value per key
- Assigning to a new key adds an entry
- Keys can be any immutable object, not just strings
- Dictionaries can be used as "records"
- Dictionaries can be used for sparse matrices

Tuples and files

Tuples: like lists, but immutable (cannot be changed)

Tuples and files

Tuples: like lists, but immutable (cannot be changed)

```
emptyT = ()
T1 = (1, 2, 3)
x = T1[1]
n = len(T1)
```

Tuples and files

Tuples: like lists, but immutable (cannot be changed)

```
emptyT = ()
T1 = (1, 2, 3)
x = T1[1]
n = len(T1)
```

Files: objects with methods for reading and writing to files

Tuples and files

Tuples: like lists, but immutable (cannot be changed)

```
emptyT = ()
T1 = (1, 2, 3)
x = T1[1]
n = len(T1)
```

Files: objects with methods for reading and writing to files

```
fil = open('myfile', 'w')
fil.write('hello file\n')
fil.close()
```

Tuples and files

Tuples: like lists, but immutable (cannot be changed)

```
emptyT = ()
T1 = (1, 2, 3)
x = T1[1]
n = len(T1)
```

Files: objects with methods for reading and writing to files

```
fil = open('myfile', 'w')
fil.write('hello file\n')
fil.close()


f2 = open('myfile', 'r')
s = f2.readline() # 'hello file\n'
t = f2.readline() # ''
```

(*Learning Python*, chapter 7)

if tests

```
course = 'icl'
if course == 'icl':
    print 'Miles / Steve'
    print 'Semester 1'
elif course == 'dil':
    print 'Phillip'
    print 'Semester 2'
else:
    print 'Someone else'
    print 'Some semester'
```

if tests

```
course = 'icl'
if course == 'icl':
    print 'Miles / Steve'
    print 'Semester 1'
elif course == 'dil':
    print 'Phillip'
    print 'Semester 2'
else:
    print 'Someone else'
    print 'Some semester'
```

- **Indentation determines the block structure**
- Indentation enforces readability
- Tests after `if` and `elif` can be anything that returns True/False

(*Learning Python*, chapter 9)

while loops

A while loop keeps iterating while the test at the top remains True.

```
a = 0
b = 10
while a < b:
    print a
    a = a + 1
```

while loops

A while loop keeps iterating while the test at the top remains True.

```
a = 0
b = 10
while a < b:
    print a
    a = a + 1

s = 'icl'
while len(s) > 0:
    print s
    s = s[1:]
```

(*Learning Python*, chapter 10)

for loops

`for` is used to step through any sequence object

```
l = ['a', 'b', 'c']
for i in l:
    print i
```

for loops

`for` is used to step through any sequence object

```
l = ['a', 'b', 'c']
for i in l:
    print i

sum = 0
for x in [1, 2, 3, 4, 5, 6]:
    sum = sum + x
print sum
```

for loops

`for` is used to step through any sequence object

```
l = ['a', 'b', 'c']
for i in l:
    print i

sum = 0
for x in [1, 2, 3, 4, 5, 6]:
    sum = sum + x
print sum
```

`range()` is a useful function:

```
range(5) = [0, 1, 2, 3, 4]
range(2, 5) = [2, 3, 4]
range(0, 6, 2) = [0, 2, 4]
```

for loops with style

Do something to each item in a list (eg print its square)

```
l = [1, 2, 3, 4, 5, 6, 7, 8]  #  or l = range(1,9)

# one way to print the square
for x in l:
    print x*x
```

for loops with style

Do something to each item in a list (eg print its square)

```
l = [1, 2, 3, 4, 5, 6, 7, 8]  #  or l = range(1,9)

# one way to print the square
for x in l:
    print x*x

# another way to do it
n = len(l)
for i in range(l):
    print l[i]*l[i]
```

for loops with style

Do something to each item in a list (eg print its square)

```
l = [1, 2, 3, 4, 5, 6, 7, 8]  #  or l = range(1,9)

# one way to print the square
for x in l:
    print x*x

# another way to do it
n = len(l)
for i in range(l):
    print l[i]*l[i]
```

Which is better?

for loops with style

Do something to each item in a list (eg print its square)

```
l = [1, 2, 3, 4, 5, 6, 7, 8]  #  or l = range(1,9)

# one way to print the square
for x in l:
    print x*x

# another way to do it
n = len(l)
for i in range(l):
    print l[i]*l[i]
```

Which is better?

The top one... Iterate directly over the sequence, try to avoid using counter-based loops...

Example: intersecting sequences

The intersection of
`['a', 'd', 'f', 'g']` and `['a', 'b', 'c', 'd']`
is `['a', 'd']`

Example: intersecting sequences

The intersection of
`['a', 'd', 'f', 'g']` and `['a', 'b', 'c', 'd']`
is `['a', 'd']`

```
l1 = ['a', 'd', 'f', 'g']
l2 = ['a', 'b', 'c', 'd']
res = []
for x in l1:
    for y in l2:
        if x == y:
            res.append(x)
```

Example: intersecting sequences

The intersection of
`['a', 'd', 'f', 'g']` and `['a', 'b', 'c', 'd']`
is `['a', 'd']`

```
l1 = ['a', 'd', 'f', 'g']
l2 = ['a', 'b', 'c', 'd']
res = []
for x in l1:
    for y in l2:
        if x == y:
            res.append(x)

res = []
for x in l1:
    if x in l2:
        res.append(x)
# res = ['a', 'd']
```

Built-in, imported and user-defined functions

- Some functions are built-in, eg:

```
l = len(['a', 'b', 'c'])
```

Built-in, imported and user-defined functions

- Some functions are built-in, eg:

```
l = len(['a', 'b', 'c'])
```

- Some functions may be imported, eg:

```
import math
from os import getcwd
print getcwd()          # which directory am I in?
x = math.sqrt(9)        # 3
```

Built-in, imported and user-defined functions

- Some functions are built-in, eg:

```
l = len(['a', 'b', 'c'])
```

- Some functions may be imported, eg:

```
import math
from os import getcwd
print getcwd()          # which directory am I in?
x = math.sqrt(9)        # 3
```

- Some functions are user-defined, eg:

```
def multiply(a, b):
    return a * b
print multiply(4, 5)
print multiply('-', 5)
```

Functions in Python

- Functions are a way to group a set of statements that can be run more than once in a program.
- They can take parameters as inputs, and can return a value as output

Functions in Python

- Functions are a way to group a set of statements that can be run more than once in a program.
- They can take parameters as inputs, and can return a value as output
- Example

```
def square(x):              # create and assign function
    return x*x
y = square(5)               # y gets assigned the value 25
```

Functions in Python

- Functions are a way to group a set of statements that can be run more than once in a program.
- They can take parameters as inputs, and can return a value as output
- Example

```
def square(x):              # create and assign function
    return x*x
y = square(5)               # y gets assigned the value 25
```

- `def` creates a function object, and assigns it to a name (`square` in this case)
- `return` sends an object back to the caller
- Adding `()` after the functions name calls the function

Functions in Python

- Functions are a way to group a set of statements that can be run more than once in a program.
- They can take parameters as inputs, and can return a value as output
- Example

```
def square(x):              # create and assign function
    return x*x
y = square(5)               # y gets assigned the value 25
```

- `def` creates a function object, and assigns it to a name (`square` in this case)
- `return` sends an object back to the caller
- Adding `()` after the functions name calls the function

(*Learning Python*, chapter 12)

Intersection function

```
def intersect(seq1, seq2):
    res = []
    for x in seq1:
        if x in seq2:
            res.append(x)
    return res
```

Intersection function

```
def intersect(seq1, seq2):
    res = []
    for x in seq1:
        if x in seq2:
            res.append(x)
    return res
```

- Putting the code in a function means you can run it many times
- General — callers pass any 2 sequences
- Code is in one place, makes changing it easier (if you have to)

Local variables

Variables inside a function are *local* to that function

```
>>> def intersect(s1, s2):
...     res = []
...     for x in s1:
...         if x in s2:
...             res.append(x)
...     return res
...
>>> intersect([1,2,3,4], [1,5,6,4])
[1, 4]
>>> res
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'res' is not defined
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'x' is not defined
```

Argument passing

Arguments are passed by assigning objects to *local* names:

```
>>> def plusone(x):
...     x = x+1
...     return x
...
>>> plusone(3)
4
>>> x=6
>>> plusone(x)
7
>>> x
6
```

Passing mutable arguments

Recall that numbers, strings, tuples are *immutable*, and that lists and dictionaries are *mutable*:

```
>>> def appendone(s):
...     s.append('one')
...     return s
...
>>> appendone(['a', 'b'])
['a', 'b', 'one']
>>> l = ['a', 'b']
>>> appendone(l)
['a', 'b', 'one']
>>> l
['a', 'b', 'one']
```

But variable names are still local

```
>>> def doesnothing(l):
...     l = ['1', '2']
...
>>> l = ['a', 'b']
>>> doesnothing(l)
>>> l
['a', 'b']
```

Importing functions

Put the definition of intersect in a module (call the file `foo.py`), then you can import it:

Importing functions

Put the definition of intersect in a module (call the file `foo.py`), then you can import it:

```
from foo import intersect
# ... define lst1 and lst2
l3 = intersect(lst1, lst2)
```

Importing functions

Put the definition of intersect in a module (call the file `foo.py`), then you can import it:

```
from foo import intersect
# ... define lst1 and lst2
l3 = intersect(lst1, lst2)
```

or

```
import foo
# ... define lst1 and lst2
l3 = foo.intersect(lst1, lst2)
```

map

```
>>> counters = range(1, 6)
>>> updated = []
>>> for x in counters:
...     updated.append(x+3)
...
>>> updated
[4, 5, 6, 7, 8]
```

map

```
>>> counters = range(1, 6)
>>> updated = []
>>> for x in counters:
...     updated.append(x+3)
...
>>> updated
[4, 5, 6, 7, 8]

>>> def addthree(x):
...     return x+3
...
# map applies its first argument (a function)
# to each element of its second (a list)
>>> map(addthree, counters)
[4, 5, 6, 7, 8]
```

Anonymous functions and list comprehensions

Anonymous functions and list comprehensions

```
# lambda is a way of defining a function with no name
>>> map((lambda x: x+3), counters)
[4, 5, 6, 7, 8]
```

Anonymous functions and list comprehensions

```
# lambda is a way of defining a function with no name
>>> map((lambda x: x+3), counters)
[4, 5, 6, 7, 8]

# you can even have a list comprehension...
>>> res = [addthree(x) for x in counters]
>>> res
[4, 5, 6, 7, 8]
```

Also check out `apply`, `filter` and `reduce`

Function design

- Use arguments for the inputs, and return for outputs: try to make a function independent of things outside it
- Avoid global variables when possible
- Don't change mutable arguments if possible
- Functions should do one thing well (not do many things)
- Functions should be relatively small

Summary

- Loops: `for` and `while`
- Functions in Python: built-in, supplied in modules, user-defined
- Defining functions with `def`
- Function arguments and return values
- Variables defined in functions are local to the function
- Mutable objects can be changed in functions
- Fancier stuff: mapping functions onto sequences