Week 6 - Lecture 11

Istvan Albert
Huck Institutes for the Life Sciences
Container type: tuple

• It is created with the ()

[ ] → list
( ) → tuple

• It is like a list – but it cannot be modified

• There is no appending or deleting from it
String formatting

Allows you to format strings: placeholder(s) → operator → values
Common string patterns

• %s → formats as string

• %d → formats (and ensures) an integer

• %A.Bf → formats float of size A with B precision

More documentation → search for: python string formatting

http://www.python.org/doc/2.5.2/lib/typesseq-strings.html
Basic string patterns

```python
print "Age1 = %15s" % 12
print "Age2 = %15d" % 12
print "Age3 = %015d" % 12
print "Age4 = %.12f" % 12
print "Age5 = %015.5f" % 12
```

```
Age1 = 12
Age2 = 12
Age3 = 12
Age4 = 12.000000000000
Age5 = 000000012.00000
```
Common float patterns

```python
print "Age1 = %f" % 123.456789
print "Age2 = %.2f" % 123.456789
print "Age3 = %6.2f" % 123.456789
print "Age4 = %16.2f" % 123.456789
print "Age5 = %16.12f" % 123.456789
```

Output:

```
Age1 = 123.456789
Age2 = 123.46
Age3 = 123.46
Age4 = 123.46
Age5 = 123.456789000000
```
Advanced string patterns (optional)

```python
print "Age1 = %15s" % 12
print "Age2 = %15d" % 12
print "Age3 = %015d" % 12
print "Age4 = %.12f" % 12
print "Age5 = %015.5f" % 12
```

```
Age1 = 12
Age2 = 12
Age3 = 000000000000012
Age4 = 12.000000000000
Age5 = 000000012.00000
```
Multiple value patterns

```python
# string format %s
print "Name=%s  Age=%s" % ("John", 12)

# integer format %d (only allows integer)
print "Name=%s  Age=%d" % ("John", 12)

# float format %f (only allows floats)
print "Name=%s  Age=%f" % ("John", 12)
```

Command Output:

```
Name=John  Age=12
Name=John  Age=12
Name=John  Age=12.000000
```
Helpful shortcuts

```python
# generate a list with identical elements
print [ 1 ] * 10

# generate repeating string elements
# (shouldn't it raise a Type Error ?!)
print '---' * 10

# a little extra work to generate the % sign
print "%%%d%%%" % 100
```

```
[1, 1, 1, 1, 1, 1, 1, 1, 1]
---*---*---*---*---*---*---*
100%
```
More on Functions

• May take zero or more parameters

• May return zero or more values

• Default values may be bound to parameters

• Keyword parameters
Positional parameters

```python
def greet(name, age):
    print "Name=%s, age=%s" % (name, age)

greet("Jane", 22)
greet(32, "Joe")
```

Command Output:
```
Name=Jane, age=22
Name=32, age=Joe
```
Keyword parameters

In general try to use keyword parameters

```python
def greet(name, age):
    print "Name=%s, age=%d" % (name, age)

greet(name="Jane", age=22)
greet(age=32, name="Joe")
```
Default parameters get auto-filled if not passed in the function call
Multiple return values

```python
# vars.py

def compute(data):
    """Returns the minimum, maxima""
    minv = min(data)
    maxv = max(data)
    return minv, maxv

data = range(10)
output = compute(data)

print type(output)
print output
print output[-1]
```

Command Output

```
<type 'tuple'>
(0, 9)
9
```
Creating (writing) files

```python
# open with the 'wt' flags (write, text)
# chose the name carefully!
# you can overwrite existing files!
stream = file('data.txt', 'wt')
print type(stream)

# check your folder for the file
stream.write("Hello World!")
stream.write("Goodbye World!")
stream.close()
```

Command Output
```
<type 'file'>
```
Explicitly add a new line

```python
# open with the 'wt' flags (write, text)
# chose the name carefully!
# you can overwrite existing files!
stream = file('data.txt', 'wt')
print type(stream)

# check your folder for the file
stream.write("Hello World!\n")
stream.write(\"Goodbye World!\n")
stream.close()
```

```
<type 'file'>
```
Appending to files

```python
# open with the 'at' flags (append, text)
stream = file('data.txt', 'at')

# run this a few times then
# check the contents of the file
stream.write("Hello World!\n")
stream.write("Goodbye World!\n")
stream.close()

# or you can run this
print list(file('data.txt'))
```
Starting point
at minimum this needs to work

```python
import week4

filename = 'short-expression.txt'

rows = week4.read_tabular(filename)
vals = week4.get_column(rows, 'CH1B_MEDIAN')

# returns a list of numbers
print sum(vals)
```

8177.0
Create the module level function that is executed as shown on the next page and produces the file shown later.

**Question:** why do we pass the rows list rather than using the filename to read the file in each report function? Wouldn’t be simpler to have:

```
report(filename, colname)
```
See next page for the content of short-report.txt
File content for the homework

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Mean</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>CH1B_MEDIAN</td>
<td>48.11</td>
<td>44</td>
<td>69</td>
</tr>
<tr>
<td>CH1I_MEAN</td>
<td>356.11</td>
<td>48</td>
<td>2781</td>
</tr>
<tr>
<td>CH1I_MEDIAN</td>
<td>369.07</td>
<td>45</td>
<td>2918</td>
</tr>
</tbody>
</table>

File column headers (tab separated)

Column name, mean, minimum, maximum
Week 6 - Lecture 12

Istvan Albert
Huck Institutes for the Life Sciences
Evolution of code

1. Make it work

2. Make it right

3. Make it fast
Problem solving: divide and conquer

1. Don’t solve the entire problem at once

2. Split it into smaller problems. Solve each then combine it with the previous step.

3. The more stupefying the error → the more likely it is something really simple!
The DRY principle

DRY = Don’t Repeat Yourself

1. When you find yourself writing the same chunk twice, put it into a function that can be called with a parameter

2. It is not the time that it saves (copy/paste is easy)

3. It prevents errors, → the truly devious ones where after a minor change some parts of the program do slightly different things.

4. Cuts down on typing errors (column name typos, variable name typos etc...)
Defensive programming

1. The **real concern** will always be that your program is doing something slightly different and you **don’t know about it**

2. Minimize the “**surface**” where human error may occur. Simple variable names, fewer typos

\[ \text{avg2\_chan1\_medl2} \rightarrow \text{disaster waiting to happen} \]

You can have simple names!

Use the scoping rules to isolate the names!
The best way to graduate from beginner is to get in way over your head.

Nothing makes you better faster.

By Steven Ray from the 37 Signals blog
The missing slide on global scope

• The widest scope a global variable can have is the file (module) in which it is declared.

• Two identically named global variables in module **MODULEA** and **MODULEB** will **stay separate**

• Use **MODULEA.VARNAME** or **MODULEB.VARNAME** to access/alter the global from the other module
Reminder on scoping rules

All names are local unless

1. Not defined in the local block. Lookup will continue in the outer, enclosing scopes.

2. Declared as global

Widest scope is at file (module) level
Exercises: create the program outputs

```python
# most exercises will test
# scope -> what value is visible
# in which codeblock

x = 100

def report(value):
    print value

print x
report(200)
print x
```

```
100
200
100
```
LEVEL = 10

def report():
    LEVEL = 20
    print LEVEL

print LEVEL
report()
print LEVEL
KM 2

```python
LEVEL = 10

def report():
    global LEVEL
    LEVEL = 20
    print LEVEL

print LEVEL
report()
print LEVEL
```
This is not good programming practice – but the code is valid
```python
def report(level):
    print(level)

220+345
report
report(100)
sum([1, 2, 3])
```
```python
x = 10

def factory(y):
    def report():
        print y
        return report

print x
myfunc = factory(x+10)
print x
```
```python
x = 10

def factory(y):
    def report():
        print y
    return report

print x
myfunc = factory(x+10)
myfunc()
print x
```
```python
VALUE = 10

def factory(level):
    def report():
        global VALUE
        print VALUE + level
    return report

define:
    factory(1)
    factory(2)
    factory(3)
```
```python
VALUE = 10

def factory(level):
    def report():
        global VALUE
        print VALUE + level
    return report

factory(1)()
factory(2)()
factory(3)()
```
```python
import km

LEVEL = 1

def report():
    global LEVEL
    print LEVEL

report()
km.report()
report()
```

```python
# the km module

LEVEL = 100

def report():
    global LEVEL
    print LEVEL

report()
```
```python
import km

def report1():
    global LEVEL
    print LEVEL

LEVEL = 1
km.LEVEL = LEVEL

report1()
km.report2()
report1()
```

```python
# the km module

LEVEL = 100

def report2():
    global LEVEL
    print LEVEL
```

'python -tu C:\cygwin\home\albert\lecture12\demo.py 2>&1' >> 'python -tu C:\cygwin\home\albert\lecture12\demo.py 2>&1'
```
import km

def report1():
    global LEVEL
    print LEVEL

    LEVEL = 1
    km.report2()

    km.LEVEL = LEVEL
    km.report2()

# the km module

LEVEL = 100

def report2():
    global LEVEL
    print LEVEL
```
```python
import km

def report1():
    global LEVEL
    print(LEVEL)
    LEVEL = 1
    km.LEVEL = LEVEL
    LEVEL = 100

report1()
km.report2()
```

```python
# the km module
LEVEL = 100

def report2():
    global LEVEL
    print(LEVEL)
```