Week 4 - Lecture 7

Istvan Albert
Huck Institutes for the Life Sciences
The built-in function `range()`

```python
1  # start=0, end=10, step=1
2  print range(10)
3
4  # start, stop, step
5  print range(100, 200, 38)
```

Command Output:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[100, 138, 176]
```
import time

# module time has a function called time

start = time.time()
time.sleep(2)
end = time.time()

print 'Start', start
print 'End', end
print 'Seconds', end - start
Wrap it into a timer function

```python
import time

def timer_report(start):
    print 'Elapsed', time.time()-start, 'seconds'

start = time.time()
time.sleep(2)
timer_report(start)
```

Elapsed 2.0 seconds
Why time our processes?

1. To understand how they scale with data size

2. There are always bottlenecks that take 90% of time whereas the rest of the program is mighty fast

3. Timing allow us to indentify where these bottlenecks are
1. We will mostly use processes that scale linearly processing twice the data → takes twice the time

2. Somewhat arbitrary and empirical definition:

   small dataset < 100,000 rows
   large dataset > 10 million rows

   in between sizes depend on the problem
1. Small datasets are easy to process because:

   • Are stored entirely in memory
   • Allow access to any part of the data
   • Can be sorted, randomized and manipulated
   • Can be iterated over repeatedly

Often we can reduce a large dataset into a small one. (we will cover processing large datasets later)
Watch the output of a resource monitor

```python
import time

SIZE = 10**7

data = range(SIZE)

print 'Size', len(data)
print 'Last five', data[-5:]
time.sleep(10)
print 'Done'
```

```
Size 10000000
Last five [9999995, 9999996, 9999997, 9999998, 9999999]
Done
```
A few empirical observations

1. 10 million integers → 150 Mb memory

2. The same integers as floats → 200 Mb memory

3. The same integers as string → 400 Mb memory
1. How much time (in seconds) does it take to generate a range of numbers from 1 to 5 million and cast float on them

2. Run the program three times and report the three numbers

3. Due next Tuesday, turn in the code and the result.
A better file reader

1. Reading tabular files are extremely common but reading them can be surprisingly devious due to nested quotes and empty columns

2. There is a module makes life easy for us

**CSV**

(comma separated values)

Somewhat misleading name
it’s not just for comma separated values
all tabular file type delimited by identical separators
Why not manually extract field

1. Sometimes it may seem that doing a string split() then extract by index is easier:

   ```python
   line = "1, 2, 3, 4"
   print line.split(‘,’)[1]
   
   this does not always work properly, you may have the separator character inside a quoted value
   ```

   for small datasets use the csv module
1. The most complicated dataset sent out during week 4

2. Quite realistic: commented lines, header row, partially empty rows, uneven column counts, missing values in some columns

3. We’ll use this type of data quite frequently, if we can process this, we can process all tabular data
Readers and parsers

1. A **reader** (also called **parser**) transforms an input into a data structure.

2. A stream just returns lines as text

3. A reader takes the text returned by the stream and breaks it into pieces (using some rules) and populates a container with the pieces
Using the csv reader

```python
import csv

stream = file('data3.txt')
reader = csv.reader(stream, delimiter='\t')
rows = list(reader)

print rows
```

It’s a list where each element is another list!
List of lists

It’s a list where each element is another list!
1. Write code that prints out the header row only

2. Write code that removes the comments, the header row and the last four (empty) lines

3. Generate a list that contains the length of each row in the file (map a function that returns the lengths)

All steps can go into the same program.
Due next Tuesday
Observation on the previous homework

```python
def = [1, 2, 3]
```

def is a **reserved word** may not be used as a variable name.

Other reserved words

```python
import, print, if, etc.
```

You may **reassign** built-in function names

```python
filter = [1, 2, 3]
```

but then you cannot use the original filter anymore – same with module filenames
The robust way to remove lines is filtering.

For simplicity you can just slice for now.

Today we’ll filter the rows and columns.
Flow control – if blocks

```python
value = 10

if value > 100:
    print 'Value greater than 100'
    print 'More code in the main block'
else:
    print 'Value less than 100'
    print 'More code in the else block'

print 'Done'
```

Output:
```
Value less than 100
More code in the else block
Done
```
Nesting if statements leads to untraceable decisions.

```python
start = 100
end = 200

if start > 50:
    if end < 30:
        print '1'
    else:
        print '2'
else:
    if start > 200:
        if end > 300:
            print '3'
        else:
            print '4'
    else:
        print '5'
```
Nesting *ifs* leads to untraceable decisions

```
start = 100
end = 200

if start > 50:
    if end < 30:
        print '1'
    else:
        print '2'
else:
    if start > 200:
        if end > 300:
            print '3'
        else:
            print '4'
    print '5'
```

BAD, BAD CODE!!!
Cyclomatic complexity

- Used to measure the complexity of a program.
- It measures the number of linearly independent paths through a program's source code.

That’s not all,

it is also about where the decisions take place. In one location or over larger section of the code
Handy utility for tracking behavior:

```
sys.exit()
```

Stops the program at a given point in execution.
Allows us to print and inspect the parameters
Exiting inside a filter function

```python
import sys

def condition(x):
    print x, type(x)
    sys.exit()
    return len(x)

data = range(100)

print filter(condition, data)
```

0 <type 'int'>
Last lecture – character delimited files

We’ve been processing files that contain lines where values are separated by a certain type of character.

Comma separated

chr1,100,200,+  

Tab separated

chr1 100 200 +  

You can also have space separated rows – can be confusing
Values may also be quoted

Sometimes there are quotation marks around some values

Comma separated

"chr1",100,200,"+"

Tab separated

"chr1" 100 200 "+"

the csv module will automatically handle these for you and does the right thing - removes the single quotes
We need a new module week4

```python
import week4
filename = 'data3.txt'
rows = week4.read_tabular(filename)
print rows
```
```python
import csv

def read_tabular(filename):
    stream = file(filename)
    reader = csv.reader(stream, delimiter='\t')
    rows = list(reader)
    return rows

# comment out the lines below
# when you are using it as a module

rows = read_tabular('data3.txt')
print rows
```
def validate(row):
    """
    Checks for empty lines, lines with empty columns and lines starting with comments
    """

    # empty lines will return an empty list
    if len(row) == 0:
        return False

    # the list is not empty but the first column has no value
    if row[0] == '':
        return False

    # while there is a first column its the first character is #
    if row[0][0] == '#':
        return False

    return True
```python
def read_tabular(filename):
    stream = file(filename)
    reader = csv.reader(stream, delimiter='\t')
    rows = list(reader)
    rows = filter(validate, rows)
    return rows

rows = read_tabular('data3.txt')
print rows
```
Our code is simple again!

```
import week4

filename = 'data3.txt'
rows = week4.read_tabular(filename)

print rows
```
Gets better – it works an all such files!

```python
import week4

filename = 'gene-expression.txt'
rows = week4.read_tabular(filename)

print(rows[:10])
```
What we really need is to extract values for a column

- We have a list of lists that starts with a header list
- We want to extract data in certain column
```python
import week4

def extract(row):
    return row[0]

filename = 'data3.txt'
rows = week4.read_tabular(filename)
rows = map(extract, rows)
print rows
```

'["chrom", "chr1", "chr1", "chr1", "chr1", "chr2",
 "chr1", "chr1", "chr3", "chr1", "chr1", "chr1",
 "chr2", "chr1", "chr1", "chr1", "chr1", "chr1",
 "chr1", "chr1", "chr3", "chr3", "chr3", "chr3",
 "chr3", "chr2", "chr1", "chr1"]'
Homework 3/3

- Use the file `gene-expression.txt` to find the sum and average of the mean feature pixel intensity at wavelength 532 nm (column CH1I_MEAN)

- Print out the code for module and program and turn it in

*You can just use indexes for columns, we will cover some more details next time*