LING 300 - Topics in Linguistics:
Introduction to Programming and Text Processing for Linguists

Week 3

Basic Python 1
Notes from Assignment 2

• Whitespace is invisible and therefore tricky
  e.g. top word = 46401 instances of ‘ ’
  Can run another sed to remove this, or a one-command fix:
    \texttt{sed 's/ +/\n/g'}

• Similar, \texttt{sed '/^$/d'} works but misses lines with spaces

• \texttt{[0-9]} is all digits (doesn’t work to do e.g. \texttt{[0-100]})
Notes from Assignment 2

Quoting!

- Be very careful with quoting! And (), [], etc. Each ' requires another ' to close it, each " requires another " to close it.

- Syntax highlighting helps a lot.
Notes from Assignment 2

Quoting!

- Double quotes interpret arguments (e.g. "$1") and escapes,
  Single quotes leave them be.


- Whitespace (spaces, tabs, newlines) is interpreted as a delimiter between arguments!
  (See TLCL Ch. 7)
Notes from Assignment 2

Stream Management!

- Be aware that almost all text filter commands can accept the input file as an argument (e.g. `sed 's/sad/happy/g' input.txt`)

- Careful with `>` (write) vs. `>>` (append)

- `>` and `>>` end the stream (alternatively can use `tee`)
Better to not generate auxiliary files, e.g.:

```bash
grep love shakes.txt > lovelines.txt
wc -l lovelines.txt
```

This works, but adds cruft and obscures things later - if we come back in a day, how exactly did we get `lovelines.txt`? Once it’s created we lose the “story,” if you will. Thus piping!

```bash
grep love shakes.txt | wc -l
```
Notes from Assignment 2

- Don’t call programs like nano / less from a script: it’ll stop execution of the script until you close that instance. nano/less are not text filters like grep/sed/tr/sort/etc.
  - They can *receive* input from stdin, they just don’t pass it through to stdout

- This and all further assignments should be runnable! (don’t write the answer, write the code that generates it)
Notes from Assignment 2

“Solutions” are posted on the course website

No claim to perfection, there is no perfect “right answer”!
**Variable Types** define different sorts of data

<table>
<thead>
<tr>
<th>Numeric</th>
<th>Sequence</th>
<th>Text</th>
<th>Truthy</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>list</td>
<td>string</td>
<td>boolean</td>
</tr>
<tr>
<td>42</td>
<td>['y', 2, False]</td>
<td>'hello!'</td>
<td>True, False</td>
</tr>
<tr>
<td>float</td>
<td>tuple</td>
<td>None</td>
<td>(next week)</td>
</tr>
<tr>
<td>42.0</td>
<td>(6, 'b', 19.7)</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

*Mapping dict*
Statements are units of code that do something

Assignment (=)

\[
\begin{align*}
\text{year} & = 2020 \# \text{integer} \\
\text{mssg} & = '\text{hooray!}' \# \text{string} \\
\text{e} & = 2.71828 \# \text{float}
\end{align*}
\]
Statements are units of code that do something

**Equality Testing** (==, !=, >, <, >=, <=)

```python
>>> year != 2016
True

>>> mssg == 'howdy!'
False

>>> e <= 3
True
```
Statements are units of code that do something

Arithmetic (+, -, *, /, **)  

>>> year * 3  
6060

>>> 'hip hip ' + mssg  
'hip hip hooray!'

>>> e / 2  
1.35914
Statements are units of code that do something

Incrementing (arithmetic plus assignment)

```python
>>> year += 18
>>> year
2038

>>> mssg *= 5
>>> mssg
'hooray!hooray!hooray!hooray!hooray!'```
Functions take input, do some computation, produce output

**Important Built-ins 1**

- `print(x)` # print representation of x
- `help(x)` # detailed help on x
- `type(x)` # return type of x
- `dir(x)` # list methods and attributes of x
  (methods are functions bound to objects)
  (attributes are variables bound to objects)
Functions take input, do some computation, produce output

Important Built-ins 2

sorted(x) # return sorted version of x

min(x), max(x) # mathematical operations
sum(x) # on sequences

int(x), float(x), bool(x) # 'casting', a.k.a.
list(x), tuple(x), str(x) # type conversion
Functions take input, do some computation, produce output

**Defining New Functions**

```python
def my_function(arg1, arg2, arg3):
    # all my amazing
    # code goes here
    return 42
```

- `def` keyword
- Function name: `my_function`
- Arguments: `arg1`, `arg2`, `arg3`

Body

- Indented one level

```python
    def my_function(arg1, arg2, arg3):
        # all my amazing
        # code goes here
        return 42
```
Control Flow organizes the order code executes

**Conditionals - if, elif, else -** enter section if condition is met

```python
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...    print('Negative!')
... elif x == 0:
...    print('Zero!')
... else:
...    print('Positive!')
Positive!
```
Control Flow organizes the order code executes

Loops - for ... in - loop over items of a sequence

```python
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...      print(w, len(w))
...  
cat 3
window 6
defenestrate 12
```
Control Flow organizes the order code executes

Loops - for ... in - loop over numbers by using range

>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
Control Flow organizes the order code executes

Loops - \texttt{for ... in} - for reading lines in a file with \texttt{open}

\begin{verbatim}
>>> for line in open('shakes.txt'):
...    print(line)
1609

THE SONNETS

by William Shakespeare
\end{verbatim}
Control Flow organizes the order code executes

Loops - while - loop until condition is met

```python
>>> # Fibonacci: sum of two elements defines the next
... a, b = 0, 1
>>> while a < 10:
...     print(a, end=' ')
...     a, b = b, a+b
... print('')
... print('')
... 0 1 1 2 3 5 8
```
Whitespace is obligatory for demarcating code blocks

The body of function definitions and control flow elements must be indented by one level

Recommended to be \t\t\t one tab . . . . or four spaces

```python
def run_tests(func, tests):
    print(f'\tRunning {} tests on the `{}` function...'.format(*tests))
    errors = 0
    for val, ret in tests:
        try:
            if type(val) == tuple:
                assert func(*val) == ret
            else:
                assert func(val) == ret
        except AssertionError:
            print(f'\terror for input {val}.format(val))
            errors += 1
        if errors == 0:
            print('\tAll tests passed!')
```
**Whitespace** is obligatory for demarcating code blocks

- Most text editors deal with whitespace semi-intelligently

- E.g., emacs sees that a file ends in .py, and interprets the text as python code (syntax highlighting) and tries to make the whitespace consistent

- Pressing the [Tab] key will jump to the logical indent. But be careful e.g. closing control flow statements, try pressing [Tab] multiple times.
String and List Indexing

>>> job_title = 'LINGUIST'

Syntax: sequence[start:end]

<table>
<thead>
<tr>
<th>Char (or List Item)</th>
<th>L</th>
<th>I</th>
<th>N</th>
<th>G</th>
<th>U</th>
<th>I</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Index</strong></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td><strong>Reverse Index</strong></td>
<td>-8</td>
<td>-7</td>
<td>-6</td>
<td>-5</td>
<td>-4</td>
<td>-3</td>
<td>-2</td>
<td>-1</td>
</tr>
</tbody>
</table>

>>> job_title[3:-1]
'GUIS'   # inclusive of start, not inclusive of end

>>> job_title[:5]
'LINGU'   # can leave off start or end
**String Methods** are functions associated with string objects

**strip, rstrip, lstrip**

```python
>>> s = '  my sTrInGggg!
' >>> s = s.strip()
>>> s
'my sTrInGggg!' >>> s = s.strip('!').strip('g')
>>> s
'my sTrInG'
```

**find**

```python
>>> s = s.strip('g')
>>> s
'my sTrInG'
>>> s.find('str')
3
```

**replace**

```python
>>> s.replace('my','your')
'your string'
```

**upper, lower**

```python
>>> s = s.lower()
>>> s
'my string'
```

**startswith, endswith**

```python
>>> s.startswith('balloon')
False
```
List Methods are functions associated with list objects

append

```python
>>> x = [1, 4, 9, 16]
>>> x.append(9)
>>> x
[1, 4, 9, 16, 9]
```

index

```python
>>> x.index(4)
1
```

remove deletes the first occurrence

```python
>>> x.remove(9)
>>> x
[1, 4, 16, 9]
```

pop removes and returns the last element

```python
>>> x.pop()
9
>>> x
[1, 4, 16]
```
Strings and Lists

Strings are like sequences of characters

Key difference: lists are mutable strings are immutable

can be changed cannot be changed


String methods to convert to/from lists

split join

>>> s = 'my string' >>> ' '.join(['your','string'])
>>> s.split() 'your string'
['my', 'string']
Assignment Walkthrough

Answers are short but can be tricky!

Think *Decomposition*

   how can I break this into smaller, doable sub-problems?

Tests provided after each function! *(non-exhaustive)*
Assignment Walkthrough

You must do

    module load python/anaconda3.6

every time you login to Quest

Run the assignment with:

    python assignment.py

The assignment must run when you are done!