Lesson 1: Getting Started with IDLE, print, Strings, Variables

Fundamentals of Text Processing for Linguists
Na-Rae Han
Objectives

- Introduction to Python 2.7
  - Try out Python IDLE (Interactive Development Environment)

- Learn Python basic syntax
  - print command
  - Strings
  - Variables

- Learn to get around in Python IDLE
  - Restarting IDLE session
  - Saving IDLE session into a file
  - Configure IDLE command-history key shortcuts
Your first code

- Type up into your IDLE shell window, followed by ENTER:
  
  ```python
  print 'Hello, world'
  ```

- What you get:

  ```
  >>> print 'Hello, world'
  Hello, world
  >>>
  ```

- Try it again.

- This time, print out 'Hello, world!!!!'
Every command you type in is stored in a buffer as command history.

You can quickly bring up and edit a previously entered command using these shortcuts:

- Mac: Ctrl+p / Ctrl+n (previous/next)
- Windows: Alt+p / Alt+n

```python
>>> print 'Hello, world!'
Hello, world!
>>> print 'Hello, worl'
```

You can edit command line
Customize command history keys

- Default IDLE key bindings assigned to previous- and next-command: Ctrl+p and Ctrl+n (Alt key in Win)
- These are hard to use! Let's change them to:
  - ⤦ (Up arrow: previous command)
  - ⤧ (Down arrow: next command)

Instructions:
1. From the menu go to "Options → Configure IDLE"
2. Click "Keys" tab
3. Scroll down and click on the line starting with "history-next". Click button "Get New Keys for Selection"
4. Scroll down to find "Down Arrow" and click on it.
5. The new key is now set to "<Key-Down>". Press OK.

Using arrows is a common practice
6. You are prompted to name your custom key scheme. Give any name.

7. Now repeat above process for "history-previous". But this time select "Up Arrow".
More commands

- Try:
  - `print 'Hello', 'world!'`
  - `4*17`
  - `print 4*17`
  - `4 * 17 + 2`
  - `4*17+2`
  - `(4 * 17 + 2) / 10`
  - `10 / 3`
  - `10.0 / 3`
```python
>>> print 'Hello', 'world!
Hello world!

>>> 4*17
68

>>> print 4*17
68

>>> 4 * 17 + 2
70

>>> 4*17+2
70

>>> (4 * 17 + 2) / 10
7
```
>>> 10 / 3
3

>>> 10.0 / 3
3.3333333333333335

Integer division: 
An integer is returned

Floating point number division: 
A floating point number (rounded) is returned
The print command

```python
>>> print 'Hello, world!'
Hello, world!

>>> print "Hello, world!"
Hello, world!

>>> print 'Hello', 'world!'
Hello world!

>>> print 'cat', 'dog', 'fox'
cat dog fox

>>> print 4500*12
54000

>>> print 'Homer makes', 4500*12, 'dollars a year.'
Homer makes 54000 dollars a year.
```
The print command

```python
>>> print 'Hello, world!'
Hello, world!
>>> print "Hello, world!"
Hello, world!
>>> print 'Hello', 'world!'
Hello world!
>>> print 'cat', 'dog', 'fox'
cat dog fox
>>> print 4500*12
54000
>>> print 'Homer makes', 4500*12, 'dollars a year.'
Homer makes 54000 dollars a year.
```
>>> print 'Hello, world!'
Hello, world!

>>> print "Hello, world!"
Hello, world!

>>> print 'Hello', 'world!'
Hello world!

>>> print 'cat', 'dog', 'fox'
cat dog fox

>>> print 4500*12
54000

>>> print 'Homer makes', 4500*12, 'dollars a year.'
Homer makes 54000 dollars a year.
Variable assignment

```python
>>> salary = 4500*12
>>> print salary
54000
```

VALUE is what’s stored, not the expression itself.
Variable assignment

```python
>>> salary = 4500 * 12
>>> print salary
54000
```

**VALUE** is what’s stored, not the expression itself.
>>> who = 'Homer'
>>> print who
Homer
>>> who
'Homer'
>>> salary = 4500 *12
>>> salary
54000
>>> print who, 'makes', salary, 'dollars a year.'
Homer makes 54000 dollars a year.
>>> who = 'Mr. Burns'
>>> salary = 50000*12
>>> print who, 'makes', salary, 'dollars a year.'
Mr. Burns makes 600000 dollars a year.
Printing vs. Returning

Prints string content

>>> who = 'Homer'

>>> print who
Homer

>>> who
'Homer'

>>> salary = 4500 *12

>>> salary
54000

>>> print who, 'makes', salary, 'dollars a year.'
Homer makes 54000 dollars a year.

>>> who = 'Mr. Burns'

>>> who = 'Mr. Burns'

>>> salary = 50000*12

>>> salary
600000

>>> print who, 'makes', salary, 'dollars a year.'
Mr. Burns makes 600000 dollars a year.

Prints string content

Evaluates expression, returns the value. Value is a string: note the quotes!
Changing variable value

Variable value is declared and then changed

```python
>>> who = 'Homer'
>>> print who
Homer
>>> who
'Homer'
>>> salary = 4500 * 12
>>> salary
54000
>>> print who, 'makes', salary, 'dollars a year.'
Homer makes 54000 dollars a year.

>>> who = 'Mr. Burns'
>>> salary = 50000*12
>>> print who, 'makes', salary, 'dollars a year.'
Mr. Burns makes 600000 dollars a year.
```
Try:

greet = 'Hello,
nameis = 'my name is'
name = 'Homer Simpson'
print greet, nameis, name

### change name to 'Abraham Lincoln'
### then print again

### change greet to 'Good morning everyone,'
### then print again

Same command. Use command history ↑ and ↓!
Variables, variables

```python
>>> greet = 'Hello,'
>>> nameis = 'my name is'
>>> name = 'Homer Simpson'
>>> print greet, nameis, name
Hello, my name is Homer Simpson

>>> name = 'Abraham Lincoln'
>>> print greet, nameis, name
Hello, my name is Abraham Lincoln

>>> greet = 'Good morning everyone,'
>>> print greet, nameis, name
Good morning everyone, my name is Abraham Lincoln
```
Can *anything* be a variable name?

- **Answer:** NO.
- **You can use:**
  - Any letter
  - Number, but **not initially**. ex: `name1` is good, `1name` is not
  - Underscore `"_"`. ex: `student_name`
- **Variable names are **case-sensitive**!**
  - `greet` and `Greet` are two different variables.
- **Following are built-in Python keywords** and cannot be used as a variable name:
  ```plaintext
  and assert break class continue def del elif else except exec finally for from global if import in is lambda not or pass print raise return try while yield
  ```
Restarting shell

- **Ctrl+F6** restarts IDLE shell

```python
>>> greet = 'Hello,'
>>> nameis = 'my name is'
>>> name = 'Homer Simpson'
>>> print greet, nameis, name
Hello, my name is Homer Simpson
>>> name = 'Abraham Lincoln'
>>> print greet, nameis, name
Hello, my name is Abraham Lincoln
>>> =============================== RESTART ===============================
>>> print greet, nameis, name
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
      print greet, nameis, name
NameError: name 'greet' is not defined
```

After RESTART, the 3 variables are no longer defined!!
String vs. number: '+\)' and \texttt{len()}\)

>>> 4 + 15
19
>>> '4 + 15'
'4 + 15'
>>> '4' + '15'
'415'
>>> 'ab' + 'cde'
'abcde'
>>> \texttt{len('abcde')}\)
5
>>> \texttt{len('4' + '15')}\)
3
>>> \texttt{len('ab' + 'cde')}\)
5
>>> \texttt{len(4 + 15)}

Traceback (most recent call last):
  File "<pyshell#12>" , line 1, in <module>
    \texttt{len(4 + 15)}
TypeError: object of type 'int' has no \texttt{len}()
String vs. number: `+' and `len()`

```python
>>> 4 + 15
19
>>> '4 + 15'
'4 + 15'
>>> '4' + '15'
'415'
>>> 'ab' + 'cde'
'abcde'
>>> len('abcde')
5
>>> len('4' + '15')
3
>>> len('ab' + 'cde')
5
>>> len(4 + 15)
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    len(4 + 15)
TypeError: object of type 'int' has no len()
```

ERROR: Integers do not inherently have length!
len() returns the length of a string measured in # of characters

>>> 4 + 15
19
>>> '4 + 15'
'4 + 15'
>>> '4' + '15'
'415'
>>> 'ab' + 'cde'
'abcde'

>>> len('abcde')
5
>>> len('4' + '15')
3
>>> len('ab' + 'cde')
5

>>> len(4 + 15)
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    len(4 + 15)
TypeError: object of type 'int' has no len()
+: number vs. string operation

>>> 4 + 15
19

>>> '4 + 15'
'4 + 15'

>>> '4' + '15'
'415'

4 and 15 are integers. 
+: *addition* operator

'4' and '15' are strings of letters. 
+ is a *concatenation* operator

>>> 'ab' + 'cde'
'abcde'

>>> len('abcde')
5

>>> len('4' + '15')
3

>>> len('ab' + 'cde')
5

>>> len(4 + 15)

Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    len(4 + 15)
TypeError: object of type 'int' has no len()
+ vs. print, * for numbers vs. strings

- Try:

  'police' + 'man'
  print 'police', 'man'
  print 'police' + 'man'

  14 * 3
  'abc' * 3
  'x' * 10
  '14' * 3
>>> 'police' + 'man'
'policeman'

>>> print 'police', 'man'
police man

>>> print 'police' + 'man'
policeman

prints **two strings**, separated by a space

prints the concatenated output as a **single string**
+, *: overloaded operators

Integer arguments. *: multiplication operator

String argument on left. * is a multi-copy concatenation operator

Just like +, * also behaves differently for number and string arguments: overloaded operators
+ as concatenation operator

```python
>>> w1 = 'school'
>>> w2 = 'bus'
>>> w1+w2
'schoolbus'
>>> print w1, w2
school  bus
>>> print w1+w2
schoolbus
>>> w1+ ' ' + w2
'school bus'
>>> print w1+ ' ' + w2
school bus
>>> [w1, w2]
['school', 'bus']
```
A string can contain a space

```python
>>> w1 = 'school'
>>> w2 = 'bus'
>>> w1+w2
'schoolbus'
>>> print w1, w2
school bus
>>> print w1+w2
schoolbus
```
A list of strings

```python
>>> w1 = 'school'
>>> w2 = 'bus'
>>> w1+w2
'schoolbus'
>>> print w1, w2
school bus
>>> print w1+w2
schoolbus
>>> w1+ ' ' +w2
'school bus'
>>> print w1+ ' ' +w2
school bus
>>> [w1, w2]
['school', 'bus']
```

A list containing two strings. 
Square bracket [] marks the list data type.
String

- String: a single piece of text, composed of a sequence of characters.
  - Enclosed in ' ' or " "

```python
>>> print "It's me, Homer."
It's me, Homer.
>>> print 'Homer says "hi."'
Homer says "hi."
```
String

- String: a single piece of text, composed of a sequence of characters.
  - Enclosed in ' ' or " "
  ```python
  >>> print "It's me, Homer."
  It's me, Homer.
  >>> print 'Homer says "hi."'
  Homer says "hi."
  ```
  - Enclosed in """ """ (triple double quotes) ← Can be multi-line!
  ```python
  >>> print """This is a long string that spans three lines."""
  This is a long string that spans three lines.
  ```
The trouble with quotation marks

But how to print:

```python
>>> print "Both ' and " are ok."
Both ' and " are ok.
```

Method 1: Use triple double quotes """".

```python
>>> print """"Both ' and " are ok."""
Both ' and " are ok.
```

Method 2: Use `backslash \` to "escape".

```python
>>> print 'Both \ and " are ok.'
Both ' and " are ok.
>>> print "Both ' and \" are ok."
Both ' and " are ok.
```
Escaping with backslash \\ 

```python
>>> print 'It\'s snowing.'

>>> print 'lala\tlala'

>>> print 'lala\nlala'

>>> multi = """This is a 
long string across 
multiple lines."""

>>> print multi

>>> multi
```

1/8/2014
Escaping with backslash \\

```python
>>> print 'It\'s snowing.'
It's snowing.
>>> print 'lala\tlala'
lala   lala
>>> print 'lala\nlala'
lala
lala
>>> multi = """This is a
long string across
multiple lines."""
>>> print multi
This is a
long string across
multiple lines.
>>> multi
'This is a
\nlong string across\nmultiple lines.'
```
Special character and \ 

- Prefixing **backslash \ (=escaping)** turns:
  - A **special character** into a normal one

<table>
<thead>
<tr>
<th>SPECIAL</th>
<th>NORMAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>' Single quote</td>
<td>'</td>
</tr>
<tr>
<td>&quot; Double quote</td>
<td>&quot;</td>
</tr>
<tr>
<td>\ Escape marker</td>
<td>\</td>
</tr>
</tbody>
</table>

- A normal character into a **special one**

<table>
<thead>
<tr>
<th>NORMAL</th>
<th>SPECIAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>n Alphabet n</td>
<td>\n</td>
</tr>
<tr>
<td>t Alphabet t</td>
<td>\t</td>
</tr>
</tbody>
</table>
Compose `print` commands to output the following:

1. Look in F:\temp\note.
2. Lisa said 'Look in F:\temp\note'.
3. Lisa said 'Look in F:\temp\note and find "my.txt"'.
4. Roses are red,
   Violets are blue,
   Sugar is sweet.
Compose `print` commands to output the following:

```python
>>> print 'Look in F:\temp\note.'
Look in F:\temp\note.
>>> print 'Lisa said \'Look in F:\temp\note\'.
Lisa said 'Look in F:\temp\note'.
>>> print 'Lisa said \'Look in F:\temp\note and find "my.txt"\'.
Lisa said 'Look in F:\temp\note and find "my.txt"'.

>>> print 'Roses are red,\nViolets are blue,\nSugar is sweet.'
Roses are red,
Violets are blue,
Sugar is sweet.
```

2 minutes
Save IDLE shell session

- You can save your IDLE shell session into a text file
  - Ctrl+Shift+s
  - Every command you typed in and its output is saved into a file
  - You should save it as a text file (.txt) and not as a python script (.py or .pyw).
    - An interactive IDLE session itself is NOT a python script! You cannot run this file!!
  - You can open the file in IDLE or any text editor program
    - Handy for reviewing later.
    - Will be required for exercise assignments.
Looking ahead

- `.split()`
  - splits a **string** into a **list**.

```python
>>> chom = 'colorless green ideas sleep furiously'
>>> chom.split()
['colorless', 'green', 'ideas', 'sleep', 'furiously']
```

```python
>>> abc = 'a-b-c-d-e-f-g'
>>> abc.split('-')
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

```python
>>> rhyme = 'eeny, meeny, miney, mo'
>>> rhyme.split(',', ')
['eeny', 'meeny', 'miney', 'mo']
```
Wrap-up

- If you want to try more commands, visit:
  - A Beginner's Python Tutorial, Lesson 2 & 3
    - [http://www.sthurlow.com/python/lesson02/](http://www.sthurlow.com/python/lesson02/)
    - [http://www.sthurlow.com/python/lesson03/](http://www.sthurlow.com/python/lesson03/)

- Exercise #1
  - Due Tuesday midnight