# Python Tutorial

## Notes

7.36/7.91/BE.390/BE.490  
Spring 2006

Notes by Graham Ruby  
except as indicated*

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>1</td>
</tr>
<tr>
<td>Environments</td>
<td>2-3</td>
</tr>
<tr>
<td>Immutable Data Types</td>
<td>3-5</td>
</tr>
<tr>
<td>Numbers</td>
<td>3-4</td>
</tr>
<tr>
<td>Strings</td>
<td>4-5</td>
</tr>
<tr>
<td>Booleans</td>
<td>5</td>
</tr>
<tr>
<td>Exceptions</td>
<td>6</td>
</tr>
<tr>
<td>Functions</td>
<td>7-8</td>
</tr>
<tr>
<td>Modules</td>
<td>9</td>
</tr>
<tr>
<td>File-Handling</td>
<td>10</td>
</tr>
<tr>
<td>Objects</td>
<td>11</td>
</tr>
<tr>
<td>Mutable Data Types</td>
<td>11-12</td>
</tr>
<tr>
<td>Lists</td>
<td>11</td>
</tr>
<tr>
<td>Dictionaries</td>
<td>12</td>
</tr>
<tr>
<td>Conditional Statements</td>
<td>13</td>
</tr>
<tr>
<td>Loops</td>
<td>13</td>
</tr>
<tr>
<td>Athena Basics (supp.)*</td>
<td>14</td>
</tr>
<tr>
<td>Emacs on Athena*</td>
<td>15</td>
</tr>
</tbody>
</table>
Linux on Athena

You will interact with your python scripts through the command line. Upon logging in to a Linux-based Athena cluster machine, you will see a prompt:

`athena %`

You can type commands at the prompt to change directories, copy files, delete files, and much more. Here are a few useful commands:

`pwd` — discover what directory you are currently in
`ls` — list the contents of your current directory
`mkdir` — create a directory with the name you provide
`cd` — move into the indicated directory
`cd ..` — move out of your current directory
`mv old new` — change the name/location of a file
`cp old new` — copy a file
`rm` — delete the indicated file

For more, see "Athena Basics" attached sheet.

Your python scripts will be created in a text editor. Emacs is one such editor. You can create or open a file by typing:

```
athena % emacs filename
```

For help with Emacs, see the "Emacs on Athena Reference Card" attached. The Emacs tutorial is recommended; open it by opening Emacs and typing Control-h t.

Run your python scripts by typing:

```
athena % python filename
```
A *python script* is a *text file* which provides a set of instructions for the *python interpreter*.

When you execute your python script, the interpreter will immediately construct an environment (the "global" environment) in the memory of your computer. An *environment* is a table which binds *variable names* (keys) to the pieces of *data* (values) that they represent. Much of the code you write will simply add to or modify the contents of that table.

```
python script:

```

```
global environment:

```

```
\[
\begin{array}{ll}
a & = 1 \\
b & = 2 \\
c & = a + b
\end{array}
\]
```

```
\[
\begin{array}{ll}
\text{generates} & \rightarrow \\
a & \rightarrow 1 \\
b & \rightarrow 2 \\
c & \rightarrow 3
\end{array}
\]
```

"=" binds the variable name to the left of it to the value to the right of it. The code to the right of the "=" will be evaluated prior to binding. Thus, on the third line of code, the value bound to "a" will be added to the value bound to "b", and the resulting sum will be bound to "c".
Variables can be re-bound to new values:

- **script**
  - a = 7
- **environment**
  - old binding is replaced by new binding
  - a ....... \* 7

The values to which variables are bound can be checked with a print statement:

- **script**
  - a = 1
  - b = 2
  - c = a + b
  - a = 7
- **environment**
  - a ....... \* 7
  - b ....... 2
  - c ....... 3
- **command line**
  - athena% python script
  - 7

print a looks up "a" in the environment... and prints its value to the command line.

### Immutable Data Types
- values cannot be modified (more on this in session II)
  - numbers, strings, booleans, functions

**Numbers**: We will deal with two types - integers (ints) and floating point decimals (floats).

- creating an `int`: a = 1
- creating a `float`: a = 1.0
- converting an `int` to a `float`: a = float(1) \ a .... 1.0
- converting a `float` to an `int`: a = int(1.999) \ a .... 1
  - does NOT round: truncates decimals!
Arithmetic operations will only return an \underline{int} if both input values are \underline{ints}; a mixture of \underline{ints} and \underline{floats} will return a \underline{float}.

\[ 2.0 \div 3 \rightarrow 0.666...7 \]
\[ 2 \div 3 \rightarrow 0 \]
\[ 2 \div 3.0 \rightarrow 0.666...7 \]

\textbf{Strings}

Can be defined using a matching pair of either single or double quotes.

\[ x = 'hello' \]
\[ y = "world" \]

Can be concatenated with "\(+\)"

\[ x + y \rightarrow 'hello world' \]

\textbf{Characters in strings are indexed starting with 0.}

\[ \text{string: } 'life' \]
\[ \text{index: } 0 \quad 1 \quad 2 \quad 3 \]
\[ x = 'life' \]

extract a letter: \[ x[1] \rightarrow 'i' \]

extract a substring: \[ x[1:3] \rightarrow 'if' \]

\begin{itemize}
\item \text{index of first character} \ \text{index after last character}
\end{itemize}
example: say 'ahhh'

1. Use double quotes to define string boundaries; single quotes will be treated as characters:
   \[ x = "say 'ahhh'" \]

2. Tell Python to treat single quotes as characters by preceding them with backslashes:
   \[ x = 'say \'ahhh\' \]

Special characters:
- \t .......... tab
- \n .......... newline

Script

print 'hi 
there'

---

Command line

`@ena % python script`

`hi`

`there`

`@ena %`

Booleans - True, False

\[ x = \text{False} \]

\[ x = 4 = = 3 \]

Double equals tests for equality

<table>
<thead>
<tr>
<th>[ 4 = = 3 ]</th>
<th>[ 4 &gt; 3 ]</th>
<th>[ 4 &lt; 3 ]</th>
<th>[ 4 &gt;= 3 ]</th>
<th>[ 4 &lt;= 3 ]</th>
<th>[ 4 &lt; 3 ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>\rightarrow \text{False} \rightarrow \text{True} \rightarrow \text{False} \rightarrow \text{True} \rightarrow \text{False} \rightarrow \text{raises an exception}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
An **Exception** is raised if an error has occurred in the execution of a script. Some ways that errors can occur:

- Giving a function/operation inappropriate or incompatible data:
  
  \[
  \text{int('hello')} \rightarrow \text{there is no integer that corresponds to 'hello'}
  \]

  \[
  4 > \text{'hello'} \rightarrow \text{the values of an int and a string cannot be quantitatively compared}
  \]

- Referencing a variable which has not been bound to any value in the environment (an **unbound variable**):
  
  \[
  a = q + 1
  \]

  \[
  \text{an exception will be raised if the variable 'q' is unbound}
  \]

If an exception is raised, execution of the script will halt and Python will print a traceback to the command line, indicating what type of error occurred and on which line of your code.
Functions (aka lambdas)

Functions take in values called arguments, perform some operation on them, and return some value.
(Note: functions can take in any number of arguments, including zero; also, they need not return a value).

Functions can be defined using `def`:

```python
def adder (x, y):
    z = x + y
    return z
```

the body of the function
is indented

```
a = adder (3, 4)  returns  7
a = adder (2, 3)  returns  5
```

When a function is executed, it will generate a new environment (whose parent environment is the environment in which it was created) for the argument variables and all variables that are bound within the function.

```
global env
adder ... lambda
a ... * 5
```

```
adder env #1
x ... 3
y ... 4
z ... 7
```

```
adder env #2
x ... 2
y ... 3
z ... 5
```
The Confusion of Functions...

If the body of a function makes reference to a variable which is not defined within the body of the function, python will look for a value bound to the variable in the function's parent environment, which is the environment in which the function was created. For instance, if you define a function in the global environment which returns 'q + 3', but you don't define 'q' in your function, then python will look for a binding for the variable 'q' in the global environment. Note that it will do this when the function is called, not when it is defined. So if 'q' is initially bound to 2 in the global environment, then you define your function, then you re-bind 'q' to 3 in the global environment, then you call your function, the value 6 will be returned. If you then re-bind 'q' to 10 in the global environment, then call your function, your function will now return a value of 13...

Avoid the Confusion !!!

Make sure that all of the variables that your function uses are either explicitly passed to it as arguments or are defined within the body of the function! (wasn't that simpler?)
Modules

Many functions are provided by Python as attributes of module objects. Modules must be imported before their functions can be used. This is accomplished with import. For example, if you wanted to take the natural logarithm of a number, you could use the method \( \log(x) \) which is part of the math module:

```python
import math  # imports the math module
x = math.log(3.14)  # a reference to the math module object
```

\( \log \) is a function associated with the math module, and can be accessed like this.

Python has many more modules. Some examples are:

**os** provides functions for interacting with the operating system (deleting files, making directories, etc).

**random** provides methods for generating random numbers, taking random elements from lists, etc.

**time** provides functions for using the computer's clock to find out what time it is.

**sys** provides methods for modifying the behavior of the Python interpreter.
File-handling - create a file object with `open`

```python
f = open('filename')  # creates a file object and binds it to 'f'
f.read()   # returns a string of the file's contents
f.readlines()  # returns a list of strings, where each string
                # is a line of text from the file
f.close()   # returns nothing; closes the file
```

To write to a file, open the file like this:

```python
f = open('filename', 'w')  # add a string 'w' as the second argument
f.write(string)  # writes the string to the file
f.close()  # don't forget to close the file object.
```

`.read()`, `.readlines()`, `.close()`, `.write(string)`

are all functions which are attributes of objects of the `file` class.
Object-Oriented Programming Systems (Oversimplified)

A Class: a type of thing (like "smiley-faces")

an instance of the class "smiley-faces": 😊
another instance of the class "smiley-faces": 😊
another instance of the class "": 😊

This is a reference to a pre-existing object:

if an object is mutable, I can change its state:

😊 ⇒ 😊 ⇒ 😊 ⇒ 😊 ⇒ 😊 ⇒ 😊

(pretend all of these changes are being made to a single drawing of a smiley-face)

Mutable Data Types

Lists

\[ \text{lst}_A = [\text{'a'}, \text{'b'}, \text{'c'}] \]

\[ 0 \quad 1 \quad 2 \quad \text{index} \]

\[ \text{lst}_A[0] \quad \text{return} \quad \text{'a'} \]

\[ \text{lst}_A.append('d') \quad \text{modifies} \quad \text{lst}_A : [\text{'a'}, \text{'b'}, \text{'c'}, \text{'d'}] \]

\[ \text{lst}_B = [ [], [], [], [] ] \quad \text{a list of lists} \]

\[ \text{len(lst}_B) \quad \text{return} \quad 3 \]
Dictionary - table of key/value pairs
keys: strings or numbers; must be unique
values: any data type; may be redundant

\[ d = \text{dict}() \]  ← creates an empty dictionary
\[ d[1] = 'lonely' \]  ← adds an entry to the dictionary
\[ d['lonely'] = 2 \]  ← adds another entry
\[ d[1] = 'unity' \]  ← changes the value which is accessed by the key 1.
\[ d['more'] = \text{dict}() \]  ← creates another dictionary as a value in the outer dictionary
\[ d['more'][3] = 'funny' \]  ← adds an entry to it.

\[ d\text{.keys()} \]  ← returns a list of keys
\[ d\text{.values()} \]  ← returns a list of values

changing a value bound to a variable  ≠  mutating a value bound to a variable

\[ a = \emptyset \]  ← creates a new int object and modifies it
\[ b = a \]  ← re-binds \( a \) to it
\[ a += 1 \]  ← modifies its bound
\[ \text{print} b \rightarrow \emptyset \]  ← modifies the bound
\[ c = [] \]  ← modifies the list bound
\[ d = c \]  ← modifies its bound
\[ c\text{.append}('z') \]  ← modifies the list bound
\[ \text{print} d \rightarrow ['z'] \]  ← modifies the list bound
Conditional Statements

if <predicate>:
  <consequence>
  <cons>
  <cons>
elif <predicate>:
  <cons>
else:
  <cons>

- all uniformly-indented lines following if statement (or elif or else statements) will be executed given that the relevant predicate is True.

Loops

For

for n in [0, 2, 5]:
  <cons>

- indented

While

while <pred>:
  <cons>

- indented

- executes <cons> if <pred> is True; after execution of <cons>, re-tests <pred> and re-executes <cons> if <pred> is still true.

Sample:

n = ∅
while n < 2: — loop
  print n
  n += 1 e — allows loop to be broken
Athena Basics

**Working with Directories and Files**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pwd</code></td>
<td>Display name of current working directory</td>
</tr>
<tr>
<td><code>echo $HOME</code></td>
<td>Display name of your home directory</td>
</tr>
<tr>
<td><code>cd dir</code></td>
<td>Change current working directory to <em>dir</em> (if <em>dir</em> is specified, uses your home directory)</td>
</tr>
<tr>
<td><code>mkdir dir</code></td>
<td>Create new directory named <em>dir</em></td>
</tr>
<tr>
<td><code>rmdir dir</code></td>
<td>Remove empty directory named <em>dir</em></td>
</tr>
<tr>
<td><code>ls</code></td>
<td>List files in current directory</td>
</tr>
<tr>
<td><code>ls options file-or-dir</code></td>
<td>List specific file/directory; <em>options</em> include:</td>
</tr>
<tr>
<td></td>
<td><code>-a</code> list all files, including &quot;dotfiles&quot;</td>
</tr>
<tr>
<td></td>
<td><code>-F</code> list files showing types (in listing, <code>/ = dir, </code>*<code>= executable,</code>@` = link)</td>
</tr>
<tr>
<td></td>
<td><code>-l</code> list with long information (mode bits, owner, size, date, name, etc.)</td>
</tr>
<tr>
<td><code>cat file</code></td>
<td>Display <em>file</em> contents all at once</td>
</tr>
<tr>
<td><code>head -n file</code></td>
<td>Display first <em>n</em> lines of <em>file</em> (if <code>-n</code> omitted, uses 10)</td>
</tr>
<tr>
<td><code>tail -n file</code></td>
<td>Display last <em>n</em> lines of <em>file</em> (if <code>-n</code> omitted, uses 10)</td>
</tr>
<tr>
<td><code>more file</code></td>
<td>Display <em>file</em> one screenful at a time; once started:</td>
</tr>
<tr>
<td><code>Spacebar</code></td>
<td>move forward one screenful</td>
</tr>
<tr>
<td><code>Return</code></td>
<td>move back one screenful</td>
</tr>
<tr>
<td><code>b</code></td>
<td>go to next line containing <em>string</em></td>
</tr>
<tr>
<td><code>string</code></td>
<td>help</td>
</tr>
<tr>
<td><code>?</code></td>
<td>help</td>
</tr>
<tr>
<td><code>q</code> or <code>Ctrl-c</code></td>
<td>quit</td>
</tr>
<tr>
<td><code>cp fromfile tofile</code></td>
<td>Make a copy of <em>file/directory</em> at new location</td>
</tr>
<tr>
<td><code>mv fromfile tofile</code></td>
<td>Move (rename) <em>file/directory</em> to new location</td>
</tr>
<tr>
<td><code>delete file-or-dir</code></td>
<td>Mark file/directory for later permanent removal (file/directory renamed with prefix of <code>#</code>)</td>
</tr>
<tr>
<td><code>undelete file-or-dir</code></td>
<td>Recover file/directory marked for removal</td>
</tr>
<tr>
<td><code>lsdel</code></td>
<td>List files/directories marked for removal</td>
</tr>
<tr>
<td><code>expunge dir</code></td>
<td>PERMANENTLY remove files/dirs marked for removal in directory <em>dir</em> (if <em>dir</em> omitted, uses current <em>dir</em>)</td>
</tr>
<tr>
<td><code>purge</code></td>
<td>PERMANENTLY remove files/dirs marked for removal in home directory and all sub-directories</td>
</tr>
<tr>
<td><code>rm -i file</code></td>
<td>PERMANENTLY remove <em>file</em> immediately (-i option has command ask for confirmation; be careful!)</td>
</tr>
</tbody>
</table>

NOTE: If you accidentally remove a file using `expunge`, `purge`, or `rm`, but the file existed previously, you may be able to recover it from the `/OldFiles` directory in your home directory. (`OldFiles` is an Athena-generated backup copy of your locker).

**Creating New Files**

You typically create new files from within application programs (e.g., using the `New` or `Save` option from a pull-down menu of a program you're working in). To create a simple text file, however, you can use the following shortcut (see also the other redirection techniques on p. 9):

- `cat > file` Create a new file by directly entering lines of text;
- `lines of text...` if you make a mistake, `delete` the file (see above) and start again, or edit the file in a text editor;
- `C-d` to add text to an existing file, use `>>` rather than `>`

For more complex text files, use `Emacs` (p. 11) or `OpenOffice` (see p. 10).
Emacs on Athena
Reference Card

Version 5
September 1993
AC-43

Starting the Editor
To create a new Emacs window for editing:
emacs &
emacs/emacs &
emacs switcher &
when filename is the name of the file for editing, which may or may not exist, and where switcher can be any combination of:
-xv use reverse video
-geometry window is not default size of window
specify size and position of window
example:
emacs -xv -geometry 800x350 Emacs &
You can also start Emacs from the Text Submenu of the Tools/ Graphics menu of the dash menu bar. For more options, type
make emacs.

Exiting the Editor
C-x C-c exit (will prompt to save modified buffers)

Getting Help
Type C-h and follow the instructions. If you are a first time user, type C-h & for a tutorial.

C-h a apropos: show commands matching string
C-h b get mode-specific information
C-h s steal Help window
C-x C-h hide Help window

Searching
C-a search forward
C-e search backward
Emacs prompts for search string, then finds it incrementally as you enter it. For types with DELETE key. Valid actions:
C-e look for next occurrence
C-e search backwards
C-x search backward

C-g cancel search (go back to start)
C-x search string
C-x search current line as search string

Replacing
M-x substitute
C-x C-s interactively replace a string
span replaces with the same string
Emacs finds each occurrence and prompts for action. Valid actions:
Space replace this occurrence
\| replace this and all others
Delete cancel search (go back to start)
Replace remaining without prompting
\| replace remaining without prompting
C-x enter replace editing mode (C-M-x to exit)
\| display these options

Error Recovery
C-g cancel any command, get out
C-x undo last command
C-t redraw garbage screen
M-a recover file recover a file lost by system crash
M-a recover-buffer recover a buffer to original contents

Moving the Cursor
C-f or C-x move forward one character
C-b or C-x move back one character
M-f move forward one word
M-b move back one word
C-p or C-x move to previous line
C-n or C-x move to next line
C-a go to beginning of current line
C-e go to end of current line
C-g go to line n
C-x go to beginning of current sentence
C-m go to end of current sentence
C-s go to end of current paragraph
C-f scroll to next screen
C-b scroll to previous screen
C-p scroll screen left
C-n scroll screen right

Inserting Text
C-x C-e insert normal printing characters
C-y insert blank line
C-x C-c insert control character
C-x C-x insert text from other buffer
C-x C-l insert text from external file

Changing Case
M-x change word to uppercase
M-y change word to lowercase
M-o capitalize word
M-x change region to uppercase
M-c change region to lowercase

Transposing
C-t transpose character and previous character
C-t transpose character and next character
C-t transpose previous character

Spelling
M-x spell-word check current word
M-x spell-region check all words in region
M-x spell-buffer check all words in buffer

Fill and Line Wrap
Text mode must be on to use the fill commands.
M-x test-mode enter text mode
C-x C-f set right margin to column n
M-x auto-fill turn on off fill mode on or off
M-g fill paragraph
M-f fill region
C-x center on line with respect to fill

Minibuffer
These keys are defined in the minibuffer:
Tab complete as much as possible
Space complete up to one word
Return complete and execute
? show possible completions
C-g abort command

Marking a Region
A region is all the text between a mark and the cursor.

C-space set mark here
C-y C-x exchange point and mark
M-f set mark
M-b set paragraph mark
C-x C-p set page mark
C-x C-b set buffer mark

Deleting and Killing
Killed text is placed into the kill buffer and can be retrieved.
Deleted text is not placed into the kill buffer, thus is not removable. Forward kills include the text currently under the cursor, backward kills do not. You can use kill and yank to move text.
Delete delete previous character
C-d delete current character
M-d kill word forward
M-Delete kill word backward
C-x kill line forward
C-x C-d kill line backward
C-a kill sentence forward
C-x C-a kill sentence backward
C-w kill region
M-w copy region
M-k kill (forward) to character c
M-k kill (backward) to character c
C-y yank back (reverse last thing killed
C-y yank back last thing killed
C-y yank back next to last thing killed
C-y cycle through previous kills
MIDDLE past text from X cut buffer to mouse cursor
RIGHT copy text between Emacs and mouse cursor into X cut buffer
Shift-MIDDLE copy text between Emacs and mouse cursor into X cut buffer, and the Emacs kill ring
C-MIDDLE cut text between Emacs and mouse cursor into X cut buffer and the Emacs kill ring

Files
C-x C-f mad new file into new buffer
C-x I insert contents of another file into this buffer
C-x C-a save current buffer into same file
C-x C-s prompt to save all modified buffers
C-x C-w write buffer to specified file
M-x write-region save marked region as a file

Buffers
C-x C-b select another buffer
C-x C-b or C-x-BUFFER LIST list available buffers
C-x C-k delete a buffer

Windows
C-x C-l hide all other windows
C-x C-x "(win)" hide window the cursor is in
C-x C-x C-L split window in two horizontally
C-x C-x C-R split window in two vertically
C-x C-x C-R move to other window
C-x C-x "(win)" scroll other window up
C-x C-x C-W scroll other window down
C-x C-x C-s scroll window right
C-x C-x C- left scroll window left

Macros
C-x ( start defining keyboard macro
C-x ) end keyboard macro definition
C-x C-k execute last-defined keyboard macro
C-x C-k C-x (append to last keyboard macro
C-x C-k C-x C-k name-last-kbd-macro
name last keyboard macro

Copyright © 1987 Massachusetts Institute of Technology