<http://www.pythonlearn.com/>

# Chapter 1

Windows/DOS Commands

These are commands that may be used in the default command interpreter, cmd.exe, as well as Powershell.

* cd - display current directory. Short for "change directory".
* cd .. - move up the directory tree one level. Doing this from C:\Users\JohnSmith\Desktop will take you to C:\Users\JohnSmith, for example.
* cd %userprofile%\Desktop - takes you to the currently logged in users desktop on most versions of Windows
* cd C:\Users\JohnSmith\Desktop - takes you to the desktop on Win 7 and 8. Replace JohnSmith with your logon or profile name.
* cd C:\Documents and Settings\JohnSmith\Desktop - takes you to the desktop on Windows XP
* cd \ - brings you back to the root of the current drive
* X: - takes you to the last used location on drive X (typically, use C: or D:). It defaults to the root directory if you have not previously cd'd to anywhere on that drive.
* dir - Display a list of files and folders, very fast.
* dir /p - Display´s long lists of files and folders per "pages". Slower.
* dir /? - Shows commands for folders and subfolders in the current folder/directory.
* cls - Clear the screen
* Up arrow - display previous commands within that session. Enter to select the command or Escape to get back to the prompt
* Tab - Toggles through existing files and folders in current directory, will auto complete the typing of a file or folder name once the first letters are typed.
* dir > myfile - creates a text file named myfile containing the directory listing. To open it with Notepad, just type "Notepad myfile" at the prompt.
* exit - Closes the command window. It has the same effect that closing the window with the "X" button (at the top right corner).
* F3 - repeats previous command (i.e. if you typed "ping 10.10.10.10" and pressed Enter. It would show "ping 10.10.10.10" again without having to retype it.

Mac & GNU/Linux Commands

These are commands that may be used in a common command interpreter, bash, which is installed by default on most Mac or GNU/Linux systems, but may not apply 100% in other command shells, such as fish, or zsh for instance.

* pwd - short for "print working directory." Displays the directory you are currently in.
* cd - a quick change directory back to $HOME directory. Short for "change directory".
* cd .. - move up the directory tree one level. Doing this from /home/johnsmith/Desktop will take you to /home/johnsmith, for example.
* cd $HOME/Desktop - takes you to the currently logged in users desktop on most versions of Mac & GNU/Linux
* cd ~/Desktop - takes you to the currently logged in users desktop on most versions of Mac & GNU/Linux
* cd /home/johnsmith/Desktop - takes you to the desktop on Mac & GNU/Linux. Replace johnsmith with your own user.
* cd / - brings you back to the root of the current drive
* ls - Display a list of files and folders, very fast.
* ls | more - Displays long lists of files and folders per "pages". Slower.
* ls --help - Shows info about listing files and folders.
* ls -l - Displays a detailed list of files and folders.
* clear - Clear the screen
* Up arrow - display previous commands within that session. Enter to select the command or Ctrl-C to get back to the prompt (In some terminals the Up arrow is not correctly mapped, thus look for an alternative in your terminal documentation)
* Tab - Toggles through existing files and folders in current directory, will auto complete the typing of a file or folder name once the first letters are typed (in some terminal/shell sessions the tab is not working, please refer to the terminal/shell session documentation you are using)
* Ctrl+A - Go to the beginning of a line.
* Ctrl+E - Go to the end of a line.
* Ctrl+F - Character forward (walk in line by characters).
* Alt+F - Word forward (walk in line by words - if the shortcut is always used, you can try Alt+Shift+F or Alt+Win+F).
* Ctrl+B - Character backward (walk in line by characters).
* Alt+B - Word backward (walk in line by words - if the shortcut is always used, you can try Alt+Shift+F or Alt+Win+F)
* Ctrl+L - Clear screen. It clears all text above the actual line.
* Ctrl+D - Delete characters in current position.
* Ctrl+T - Transpose characters (change positions).
* Alt+T - Transpose words (change positions).
* Ctrl+K - Cut till the end of the line.
* Ctrl+U - Cut till the beginning of the line.
* Ctrl+W - Cut previous word.
* Alt+D - Cut next word.
* Ctrl+Y - Paste recently cut text.
* Ctrl+C - Delete whole line.

Python commands

* quit() or exit() -ends Python on Windows.
* ctrl + z - suspends python process on Mac and GNU/Linux, resume with fg command
* quit() or ctrl + D -Tells "End Of File" to Python on Mac and GNU/Linux.
* quit() or ctrl + C -Kills Python process on Mac and GNU/Linux.

Conditional Steps, repeated steps, loop

* Conditional Steps - if ... elif .. else
* if (expression):
* (command)
* elif (expression):
* (command)
* else:
* (command)
* Conditional Steps (Multi elif) - if ... elif .. else
* if x < 0:
* x = 0
* print 'negative changed to zero'
* elif x == 0:
* print 'zero'
* elif x == 1:
* print 'single'
* elif x == 2:
* print 'double'
* else:
* print 'more'
* Repeated steps - while (Don't forget to modify the variable used in the conditional, otherwise this will run forever!)
* while i < 6:

print "At the top i is %d" % i

* Repeated steps - another example

Assign a value to i which is greater than 2

while i > 2:

print "Right now i is %d" % i

i=i-1

* Loop - for
* for (y) in (x):

(command)

Atif : Reserved words rearranged in Alphabetical Order

and

as

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

with

yield

" with global yield finally in break, assert and continue for import raise or try def return. while del exec (think of Dell Executive... lol...) is not from lambda class except elif, print pass as if... else... "

**Assigning from Mathematical Expressions**

More complex mathematical expressions can be evaluated for variable assignment than just numeric literals. This can be achieved by way of the following operators:

* Addition ( + )
  + x = 5 + 10
* Subtraction ( - )
  + x = 5 - 10
* Multiplication ( \* )
  + x = 5 \* 10
* Power ( \*\* )
  + Exponentiation ( 510 )
    - x = 5\*\*10
  + The \*\* operator can also be utilized for root calculations since roots are just fraction exponents ( 10−−√ = 101/2 )
    - x = 10\*\*.5
* Division ( / )
  + x = 5 / 10
* Modulo ( % )
  + Used to return the remainder of a division instead of the quotient answer ( 10 % 3 = 1, since 3 \* 3 = 9 and 10 - 9 leaves a remainder 1 )
  + x = 10 % 3

*Nota Bene:* There is a marked difference between *integer number* division and *floating point number* division. Because Python treats these two numeric types distinctly different, you might get erroneous answers when doing division operations without precaution. (read more on **Data Types: Other Caveats** below)

**Operator Precedence (PEMDAS)**

Python also has a built operator precedence that affects the flow of mathematical evaluation. This means that certain calculations in more complex mathematical expressions will be evaluated first before moving on to others. An easy way to memorize the orders of operators is the acronym PEMDAS. Python's operator precedence is as follows:

PEMDAS:

1. Parenthesis (P)
2. Exponents / Roots (E)
3. Multiplication and Division (M D)
4. Addition and Subtraction (A S)

Example:

name = raw\_input("What is your name, human? ")

print "Hello", name + "! I am Python :D"

Note Chandrama:In Python 3.x, **input()** replaces **raw\_input()**, for input from the console.Also remember to use the **print()** function with Python 3.4, including parentheses, for output to the console, instead of the old **print** command that did not require parentheses.So the example above will be in Python 3x as follows:

name = input("What is your name, human? ")

print ("Hello", name + "! I am Python :D")

## Data Types

Notice how the mixed use of string literals and variable names in the second print line comes together in the final output. Until now, we've only seen the + operator used in the context of arithmetic addition. But Python didn't combine variable **name** + the other sentence to form a whole new value like passing in 2 + 2 into Python would produce 4. This is because Python is aware of something called "data types" and it acts on each different type accordingly.

The **data types** we have learned about so far are:

* **Numeric types**
  + **Integer numbers**
    - A whole number, i.e. a number that is not a fraction
    - Ex: 1, 5, 33, -24, 5000
  + **Floating-point numbers**
    - A number with a fraction or decimal point
    - 17.34, 98.2, -768.001, 0.2
* **Strings**
  + A collection of characters
  + This can be alphabetic characters, special characters, or even numeric characters (but saved as type string, enclosed in "quotes")
  + Ex: "Alphabetic", "ch@r@cter$!!!", "1234567890"

Nota Bene: If you are unsure of what data type a particular variable, constant, or literal is, you can ask Python by using the type() function.

Because variable **name** and the string literal passed in to print were both of variable type string, Python did something called concatenation rather than addition with the + operator. In effect, concatenation attaches the second string to the end of the first.

String concatenation example:

print "Hello" + "Coursera!"

#Output: HelloCoursera!

Note that when using concatenation with the + operator, sentence spacing isn't automatically allotted for. Whereas print "Hello", "Coursera!"would have output "Hello Coursera!". Also the multiplication \* operator can be used for multiple string concatenation:

print "Free Education!", "Hip-hip-hooray! " \* 3 + "Thank you Coursera!"

#Outputs: Free Education! Hip-hip-hooray! Hip-hip-hooray! Hip-hip-hooray! Thank you Coursera!

### Type Conversion

raw\_input is great for assigning values to variables we don't have predefined. However it comes with one caveat, all input accepted is saved as a string. Regardless of if we're specifically prompting for a numeric value and the user enters numbers only, the resulting value will be saved as type string. After all, even numeric characters are just characters, as far as Python knows, all the user typed in were a bunch of characters.

This can lead to logical errors if we tried to do any sort of arithmetic calculation with data that is essentially of type string. Best case scenario, you may end up botching your variables by concatenating or multiple concatenating them together

x = raw\_input("Enter a number: ")

y = raw\_input("Enter another number: ")

z = x + y

print "The sum of", x, "and", y, "is", z

#Output:

# Enter a number: 23

# Enter another number: 45

# The sum of 23 and 45 is 2345

Or in the worst case Python may give you an error if you try to make it do a calculation with strings that it simply cannot make sense of like division or root calculation.

Type-casting is a universal programming notion of converting the value of a variable from one type into another. In Python, this is achieved by utilizing functions such as int(), float(), or str(). If we want to do arithmetic calculation with user input values, we need to convert them into a numeric type first.

z = int(x) + int(y)

print "The correct sum actually was", z

#Output: The correct sum actually was 68

Nota Bene: You can only type-cast between strings and integers and vice versa if the characters are numeric. Any alphanumeric or special character combination will result in a Python error.

### Other Caveats

**Permanence: Data type-casting vs. Data conversion**

Converting a variable to integer or float without redefining the variable with the operand "=" will not save the variable as integer or float in memory for later use. The following example illustrates this concept.

>>> var = raw\_input('Enter Number:\n')

Enter Number:

35

>>> print type(var)

<type 'str'>

>>> print float(var)

35.0

>>> print var

35

>>> print type(var)

<type 'str'>

>>> var = float(var)

>>> print type(var)

<type 'float'>

**Truncation: Integer division vs. Floating-point division**

There is a marked difference between integer number division and floating point number division. Because Python treats these two numeric types distinctly different, you might get erroneous answers when doing division operations without precaution.

For instance, calculating 99.0/100 in Python will evaluate the decimal numbers in the answer as well, giving you a solution of 0.99.

Whereas when evaluating the two whole number division 99/100, Python doesn't recognize any decimal points and therefore truncates (not round up/down, truncates simply means "cut off" or "discard") any value after the decimal point, effectively give you an answer of 0.

As soon as one operand is a float, the rest of that expression will be evaluated as a float. Be careful however that following the order of operations, this only happens when the first float is encountered. Example 1+2\*3/4.0-5 will be evaluated as -2.5 while 1.0+2\*3/4-5 is evaluated as -3.0 because the addition operation happens much later in the order of operations. Everything before the 1.0 is encountered as an operand is evaluated as integers.

Nota Jenevain: Actually, in Python 3, both expressions evaluate to -2.5.

## Python Comments

It is good practice to document your code as your programs begin to get larger than the introductory "Hello World" level.

Code documentation in Python is done using the # symbol. Anything after a # is ignored by the Python Interpreter, meaning none of it is read or interpreted as code. Multi-line comments can be done by putting multiple lines of comments between lines with three single quotes.

These comments are for programmer's benefit only. It increases the clarity and readability of your code, whether that's you yourself re-visiting old code from months or years ago and trying to remember what you were thinking when you wrote that particular algorithm, or if it's someone else altogether looking through your code trying to make sense of your programming logic.

Consider the following undocumented code:

a = float(raw\_input('Enter a number: '))

b = float(raw\_input('Enter a number: '))

c = float(raw\_input('Enter a number: '))

x = (a + b + c) / 2

y = (x\*(x-a)\*(x-b)\*(x-c))\*\*0.5

print '%.2f' %y

Generally, all useful programs read some data, do some work with the data, and then produce results, and that is apparently happening here so it can't be faulted on that count. For all intents and purposes, this is a perfectly functional program. But it's almost impossible to distinguish what it is calculating nor what its purpose is. And even though this program successfully runs, its output isn't very clear on the purpose or solution either:

Enter a number: 23

Enter a number: 34

Enter a number: 12

66.81

Now lets look at it with a bit more careful documentation:

# Python Program to calculate the area of triangle

# Prompts the user to input the 3 sides of the triangle

a = float(raw\_input('Enter a number: '))

b = float(raw\_input('Enter a number: '))

c = float(raw\_input('Enter a number: '))

# Formula to calculate the semi-perimeter

x = (a + b + c) / 2

# Calculates the area

y = (x\*(x-a)\*(x-b)\*(x-c))\*\*0.5

# Output final area value

print '%.2f' %y

Now things are starting to take shape. Again, this doesn't affect the actual Python interpreted code in any way, but now it has become much more legible and easier to understand both what its purpose is as well as what each segment of the program is attempting to do, even if we had never seen the formula being mentioned before.

While we're at it, let's make the output a bit clearer as well:

# Python Program to find the area of triangle

# Prompts the user to input the 3 sides of the triangle

side1 = float(raw\_input('Enter first side: '))

side2 = float(raw\_input('Enter second side: '))

side3 = float(raw\_input('Enter third side: '))

# Formula to calculate the semi-perimeter

sPerim = (side1 + side2 + side3) / 2

# Calculates the area

area = (sPerim \* (sPerim - side1) \* (sPerim - side2) \* (sPerim - side3) ) \*\* 0.5

# Output final area value

print 'The area of the triangle is %0.2f' %area

'''

Sample Program Output

Enter first side: 5

Enter second side: 6

Enter third side: 7

The area of the triangle is 14.70

'''

Not only have the refined prompts and program output become much more understandable, the programmer's contextually aware variable naming scheme (see **Variable Naming** above) have practically made all of the formula self documenting.

There is no clear-cut right or wrong way to document your code or even name your variables. Those kind of decisions deal less with Python's strict language syntax rules and fall more along the lines of general programming design choices. The first code snippet shown still works perfectly fine, but a better designed and well documented Python program is considered superior. ([source attribution](http://www.programiz.com/python-programming/examples/area-triangle))

## Debugging

* Here are a few pointers on debugging.

\*Avoid spaces in between variable names. Python thinks there are two operands. Example:

>>>Madam Medousa = "lets start the bout"

SyntaxError: invalid syntax

#correct syntax

>>>madam\_medousa = "lets start the bout"

>>>print madam\_medousa

"lets start the bout"

* Common Syntax Errors. 1) SyntaxError: invalid syntax 2) SyntaxError: invalid token

\*Runtime Error: using a variable before it is defined. Example:

>>>cat= "Garfield"

>>>food = "burgers"

>>> Garfield + food

NameError: name"Garfield" is not defined

# "Garfield" is a string in the variable cat.

# Hence, Garfield is not a defined as a mnemonic variable.

\*Case sEn5iTiVity (sensitivity!) matters:

>>>Olive = 5

>>>olive = "olive oil"

>>>oLiVe = 7

# variable names are case sensitive,

# all of these variations of olive are separate variables.

# Chapter 4

## Built-In Functions

Whether we realized it or not, we have already been using functions extensively in our programs since week 1. Python has a number of built-in functions we can use without any knowledge of how its internal code actually works, so long as we know what the function name is. Examples include:

* raw\_input()
* type()
* int()
* float()
* str()
* max()
* len()

To see a full listing of Python's built-in functions, visit the [online Python documentation](https://docs.python.org/2/library/functions.html).

### Modules

These module files can then be incorporated into your code with the #import command. This gives you a whole new library of functions to utilize in your code.

### Bringing It All Together: A Practical Implementation

Dr. Chuck mentions that functions may not seem useful right now when our programs are generally only a few lines long. However, for pedagogical reasons, we are asked to re-implement our simple "Compute Pay" program to utilize a function call in our week 4 problem set. Similarly, all the lecture examples, self admitted by Dr. Chuck himself, are rather simple and silly. Even in these notes, for pedagogical reasons similar to Dr. Chuck's, I've kept all of the examples simple, leaning towards concept comprehension rather than complex implementations.

However, this kind of oversimplification brings about its own set of problems. Fellow students might wonder why bother with a product() function when they could simply use the arithmetic \* multiplication operator instead? Why write a function that only does a print execution when a straightforward print would accomplish the same trivial task?

It is equally important to get a practical perspective of code concepts than just simplified examples. At a certain point, explaining in abstract terms becomes detrimental. To that end, I've whipped up some example code that implements all of the above topics to demonstrate them in real-world application.

# Caesar Cipher - A Python program to encrypt/decrypt messages

# Python v 2.x (Not Supported in v3.x)

# Import Declaration

import string

# Function Definition

def cipher(message, shift, encrypt):

if encrypt == False : shift = 26 - shift

return message.translate(

string.maketrans(

string.ascii\_uppercase + string.ascii\_lowercase,

string.ascii\_uppercase[shift:] + string.ascii\_uppercase[:shift] +

string.ascii\_lowercase[shift:] + string.ascii\_lowercase[:shift]

)

)

# Prompt user for encryption criteria

message = raw\_input("Enter plain-text message: ")

shift = int(raw\_input("Choose your cipher shift value: "))

# Encrypt message

message = cipher(message, shift, encrypt = True)

print "Encrypted cipher-text: " + message

# Prompt user for decryption

key = int(raw\_input("Please enter cipher-key to decrypt your message: "))

# Attempt decryption

if key == shift:

message = cipher(message, shift, encrypt = False)

print "Decrypted cipher-text: " + message

else:

message = None

print "Invalid key. Your message has been deleted."

print "This program will now self destruct!"

* This throws up an error:

AttributeError: 'str' object has no attribute 'translate' on line 10

This program utilizes the principles of a classic [Caesar Cipher](http://en.wikipedia.org/wiki/Caesar_cipher) in order to encrypt and decrypt messages. The textbook [briefly covers module imports and dot notation](http://www.pythonlearn.com/html-008/cfbook005.html#toc46), however if you have only watched the lecture videos on Coursera, just ignore the verbose minutia in the latter part of the function definition and take it on faith that return message.translate() is returning an encrypted/decrypted value back to the function call.

The point to take away from this is that without the function definition, all of that [string manipulation logic](https://docs.python.org/2/library/string.html#module-string) happening would have to be implemented over and over in the code whenever the **message** variable needed to be encrypted or decrypted by the user. However, as the current implementation stands, every time **message** is manipulated, a simple cipher() call is made that only takes up one line. This drastically cleans up the code and makes debugging much less of a hassle since if you ever wanted to tweak some parameter in the algorithm, there is only one place you have to go to edit instead of several.

Function cipher() also utilizes arguments and parameters to pass in the user input **message**, the encryption **shift** value (i.e. the 'key'), and a boolvalue to determine whether to encrypt or decrypt **message** with that key.

Sample Output:

Enter plain-text message: Cryptography using Python!

Choose your cipher shift value: 17

Encrypted cipher-text: Tipgkfxirgyp ljzex Gpkyfe!

Please enter cipher-key to decrypt your message: 17

Decrypted cipher-text: Cryptography using Python!

# Chapter 5

# Lecture Notes

### "is" and "is not" Operators

"is" can be used in logical expressions.

It implies 'is exactly the same instance, type and value'.

Whereas '==' compares two items and will declare the comparison as 'True' if the items are the same type and have the same physical value, the 'is' operator only agrees they are the same if they are actually the same physical instance of the value. Because this behavior is implementation dependent, 'is' should only be used with predefined constant values such as 'True', 'False' and 'None'. '==' should be used more frequently than 'is' and 'is not' as these operators are intended for specific situations. 'is' and 'is not' are best used when testing for specific values like "None" or "True/False". '==' is best used for testing numeric values. (i.e. 'if i == 4')

# Chapter 6

String Methods: <https://docs.python.org/2/library/stdtypes.html#string-methods>

## Slicing of Strings

A slice is a character or a series of character within a string. A string slice is defined by the index operator, square brackets '[ ]'. Indexes (or subs) are integers or numeric expression staring with 0.

MySlice = sliceofdata[beginning position:'up to' position].

beginning position first position of the slice. The index starts with zero 'up to' position last position of the slice plus one or up to but not including.

# Builtin String Functions

## len

For our purpose returns the length the string.

.find():

Use the .find() function when you want to know the position of a character or a sub string within a string.

To find the position of the "3" in the string "0123456":

data="0123456"

pos=data.find('3')

print pos

>3

To find the position of "l" in "realmadrid":

word="realmadrid"

pos=word.find('l')

print pos

>3

To find the position of 'cd' in the string 'abcdef'

word = 'abcdef'

pos = word.find('cd')

print pos

>2

(Note: an error you may get is "Expected character buffer," which can happen if you omit the quotation marks from your string.)

.upper():

Use the .upper() function to capitalize each character in the given string.

data='Hello Python'

capdata=data.upper()

print capdata

>HELLO PYTHON

.replace()

Use the .replace()to search and replace a character or letter in a given string.

data='Hello Python'

rpld=data.replace("Python","Leo")

print rpld

>Hello Leo

.lower()

Use the .lower() function to convert the sting to the lower case.

data='Hello Python'

capdata=data.lower()

print capdata

>hello python

.startswith()

Use the .startswith() function to check if the string begin with that character or not.

data='Hello Python'

print data.startswith('H')

>True

.rstrip()

Use the .rstrip() function <without any parameters> to remove any additional white spaces

, ' ', from the right side of the calling string.

data1 = 'Python is cool '

data2 = 'Yay'

print data1.rstrip() + data2

>Python is coolYay

.lstrip()

Use the .lstrip() function <without any parameters> to remove any additional white spaces

, ' ', from the left side of the calling string.

greet = ' Hello Bob '

print greet.lstrip()

>'Hello Bob '

.strip()

removes both beginning and ending whitespaces of the calling string. In the above example, where greet = ' Hello Bob ':

print greet.strip()

>'Hello Bob'

.strip() also removes the new line character at the end of the string. This is something more about .strip() method.

Note that I have assigned to variable var the following value: var = 'test new line\n'

Case1: In this case we just print the value of var and note that the \n & print output a blank line

>>> print var

test new line

Case2: But, in this case if we apply .strip() to var and print the result, the blank line "disappear"

>>> print var.strip()

test new line

>>>

Case3: Another example to clarify. This time we apply strip() directly and obtain only the phrase:

>>> "this is another new line\n".strip()

'this is another new line'

# dir()

dir isn't a method in the str type in python; dir is rather a free function that can be used with any variable/type to know what methods are supported on this variable/type.

[Back to Resources](https://share.coursera.org/wiki/index.php/Pythonlearn:Resources)

# Some basic notes

To find all functions which can be used for a string, you can do following fruit = 'banana' dir(fruit) <--- this will give you all functions which can be applied on string. Or you can just do dir(str).

Immutable: When you apply function on a string (say fruit.upper()) then a new copy of original string with modifications as per function is provided. Original string remains intact.

# Additional Notes: a better (?) way to join strings with variables

Python offers a cleaner way to join variables inside strings. Follow the example

name = "Karl"

age = "30"

print name + " is " + age + " years old"

could be replaced with the much simpler (method called string interpolation)

print "%s is %d years old" % (name, age) -- % is an operator that will allow us to replace the %s with a literal or the value of a variable

Above syntax gives an "TypeError: %d format: a number is required, not str"

print "%s is %d years old" % (name, int(age)) --Added int conversion

Since Python 2.6 (and including Python 3), an interpolation method named #format is available.

sample text: "This line of {} is being used to demonstrate the #format {}.".format('text', 'method')

yields: "this line of text is being used to demonstrate the #format method.

read more here: <https://docs.python.org/2/library/string.html#format-string-syntax>

This method is powerful, flexible, and intuitive.

Can we say that "join strings with variables" is a concatenation?

I think this is a better way to concatenate variables of different data types all at the same time. Like for example in this case we are concatenating name with age and if age was a decimal value it would still work fine.

name = 'john'

age = 36

print "%s is %d years old" %(name,age)

# Chapter 7

<https://docs.python.org/2.7/library/functions.html>

## Working with files

Before Python can access file information it must "open" the file. open() is the built-in function which tells Python to open the file. open() takes two parameters 'filename' and 'mode'. If you leave the second parameter off 'open(filename)' Python will simply open the file in read mode ('r'). Opening a file does not cause Python to read all the data in the file, but it makes the information in the file available to Python to use - it creates a connection between Python and the file on the hard drive, referred to as a "Handle".

### Create and write in a file

In the lecture it's all about how to open and read a file. So here is a very simple example about how to create a file and write something in it.

The code below will read the file 'mbox.txt', and find all emails from 'umich.edu', and write these email addresses into a file named 'emailaddress.txt'

fhand = open('mbox.txt','r')

whand = open('mailaddress.txt','w')

for line in fhand:

if line.startswith('From:') and line.endswith('umich.edu\n'):#don't forget the newline '\n', otherwise you will get nothing in your new file

whand.write(line[6:])#I simply count the index number, in fact you may use 'find' or something

fhand.close()

whand.close()

A complete scenario for dealing with a file would thus look like this:

fhand = open("words.txt")

# Do whatever you like with the file contents here, like reading them through a loop.

fhand.close() # Close the file.

An even more elegant way to write the previous scenario is to use the with statement:

with open("words.txt") as fhand:

# Do whatever you want here with the file contents.

Note that there's no need to explicitly close the file here, since a with statement automatically calls close on the file handle once the last statement in its block is executed. fhand is only usable inside the with statement, that is, inside the code block indented under with.

The with statement is a python construct that can be used for some other types as well, in situations where scoped resource management is desired. The with statement entails some deeper details on objects it can be used with, and file handles satisfy those requirements to be used with the with statement. The with statement comes handy here for ensuring resources associated with file handling are managed correctly.

# Chapter 8

## Lists

Collection - allows us to put many values in a single variable.  
Simple variables are not collections. The previous value is overwritten when changed.

* A List is made up of list 'constants'. Lists are surrounded by square brackets [] and the constants in the list are separated by commas. ([2,4,6,8])
* A List element can be any Python object, even another list
* A List can be empty
* Lists are mutable (they can be changed)
  + This is a big difference between lists and strings. Strings are NOT mutable. Strings and lists are similar when it comes to other functions such as concatenation, slicing, the pattern for indexing the elements within the list, and others.
* When len() is used on a list, it counts and returns the number of constants that make up the list. (not the number of characters)
* Lists can be concatenated using +
* Lists can be sliced
* List is a unique type that can be checked using type() (result: <type 'list'>)
* An empty list can be created with list(), or setting a variable to "[]", e.g. stuff = []
* Lists can be tested for contents using in/not in
* List is an ordered sequence

### List Methods

Read more about list Methods online at: <http://docs.python.org/tutorial/datastructures.html>  
On the page there is method: list.copy(). But when I test this method, python told me: 'list' object has no attribute 'copy'

And now I realized that's a difference between python2.7 and python 3

Methods: append, count, extend, index, insert, pop, remove, reverse, sort

append() - use to add constants to the end of a list (lst.append('word'))

remove() - use to remove constants from a list

sort() - use to sort the list. It changes the list permanently.

### List Functions

len() - find length, max() - find highest value, min() - find lowest value, sum() - add all values  
average can be found with sum()/len()

### Selection in a List

Look up items in a list using an index specified in square brackets. The number in the index indicates the numbered item in the list.  
For example: For friends = ['Joseph','Glen','Sally'] using "print friends [1]" will print out "Glen" (remember numbering starts with zero)

### Using Strings and Lists Together

"Split" breaks a string into parts and produces a list of strings.

abc = 'With three words'  
stuff = abc.split()  
print stuff  
['With','three','words']  
\* Split sees multiple spaces as one, creates one split. \* Define a delimiter for split by entering a parameter (i.e. split(';') to use semicolon as a delimiter)

# Del

* Del is a command which removes what corresponds to the index of the list. For example, if we have

a = [1, 2, 3, 4, 5] del a[0] returns [2, 3, 4, 5].

If we have multiple arguments after del, then del first updates the list, and then removes the next argument. For example if

a = [1, 2, 3, 4, 5] del a[0], a[2] returns [2, 3, 5], since a[0] first gets deleted yielding [2, 3, 4, 5], then since the 2-element of [2, 3, 4, 5] is 4, 4 gets deleted.

Here is another solution for Exercise 8.1 with guarding action taking place. Critique and comment, please:

def chop(mlist): # this function returns nothing

if len(mlist)==0:

print 'there is nothing to chop'

exit()

if len(mlist)==1:

print 'only one element found, deleted..'

del mlist[0]

exit()

else:

print 'first and last element deleted'

del mlist[0] #Reduced this part of the code with del mlist[0], mlist[-1]

del mlist[len(mlist)-1]

Here is another solution for Exercise 8.1

## Mike R.'s comment: This alternate solution for the 'chop' function does not produce the requested result. The exercise specifically states to modify the original list, not a copy. That is why it says to return "None", because the original list is being changed (and it doesn't make sense to return a result in that situation). This solution is based on a misinterpretation of the exercise.

origList = [1, 2, 3, 4, 5, 6, 7]

def chop(origList):

newList = list(origList) # copy to a new list

del newList[0] # remove 1st element

del newList[len(newList)-1] # remove last element

return newList[:] # the Question doesn't make sense. Why write a function that return None.

# I changed it to return the newList[:] instead. Just comment out this line

# with a "#" and it will return None.

def middle(origList):

newList = list(origList)

return newList[1:-1]

print "Original list:", origList

print "Chopped list:", chop(origList)

print "Only middle list:", middle(origList)

### Solution to Exercise 8.2:

fhand = open('mbox-short.txt')

count = 0

for line in fhand:

words = line.split()

# print 'Debug:', words

if len(words) == 0 : continue

if words[0] != 'From' : continue

try:

count = count + 1

print words[22]

except:

print 'Index out of range.'

break

With the result:

"Index out of range."

### Solution to Exercise 8.3:

fhand = open('mbox-short.txt')

count = 0

for line in fhand:

words = line.split()

# print 'Debug:', words

if len(words) == 0 or words[0] != 'From': continue

try:

count = count + 1

print words[24]

except:

print 'Index out of range.'

break

With the result:

"Index out of range."

Here is another solution to Exercise 8.3, using the "and" logical operator, instead of "or":

fhand = open('mbox-short.txt')

count = 0

for line in fhand:

words = line.split()

if len(words) != 0 and words[0] == 'From' :

count = count+1

print words[2], count

else : continue

### Solution to Exercise 8.6:

numlist = list()

while True:

inp = raw\_input('Enter a number, when done entering numbers, type "done": ')

if inp == 'done':break

value = float(inp)

numlist.append(value)

print numlist

print 'this is: ',numlist

print min(numlist)

print max(numlist)

Result:

Enter a number, when done entering numbers, type "done": 12

[12.0]

Enter a number, when done entering numbers, type "done": 3

[12.0, 3.0]

Enter a number, when done entering numbers, type "done": 56

[12.0, 3.0, 56.0]

Enter a number, when done entering numbers, type "done": 156

[12.0, 3.0, 56.0, 156.0]

Enter a number, when done entering numbers, type "done": done

this is: [12.0, 3.0, 56.0, 156.0]

3.0

156.0

### About Exercise 8.6

the code below is not an exact solution, but an interesting way to do the same thing:

numlist = list()

while True:

inp = raw\_input("Enter a number: ")

if inp == "done":

break

num = float(inp)

numlist.append(num)

numlist.sort()

print "Maximum:",numlist[-1]

print "Minimum:",numlist[0]

# Chapter 9

List - a linear collection of ordered values.

Dictionary - an assortment of unordered labeled values, similar to a database

## Dictionaries

Different names for Dictionaries:

* Associative Arrays - Perl/Php
* Properties of Map or HashMap - Java
* Property Bag - C#/.Net

### Using Dictionary to tally/count items

The "in/not in" operators can be used to see if a key exists in the dictionary - without resulting in a traceback error.

Example (9C):

counts = dict()

names = ['csev','owen','csev','zqian','cwen']

for name in names:

if name not in counts:

counts[name]= 1

else:

counts[name] = counts[name] + 1

print counts

The above operation is so common, that the method 'get' does it for us.

print counts.get(name,0)

Performs the same operation as:

if name in counts:

return counts[name]

else:

retrun 0

So, example 9C can be simplified down to 9D, below:

Example (9D):

counts = dict()

names = ['csev','owen','csev','zqian','cwen']

for name in names:

counts[name] = counts.get(name,0) + 1

print counts

## Few more methods to iterate a dictionary using example 9C's dictionary

Iterate over keys: counts.keys():

for name in counts.keys():

print name

Iterate over the values in a dictionary: counts.values():

for occurrence in counts.values():

print occurrence

Iterate over the elements ((key, value) pairs) in a dictionary : counts.items():

for name,occurrence in counts.items():

print name, occurrence

# Chapter 10

## Tuples

Tuples are a sequence that behave much like a list, except:

* Tuples are immutable (cannot be altered)
* displayed surrounded by parentheses '(,)' rather than brackets '[,]'
* surrounding by parenthesis is just visual help; Comma is what makes tuple type
* cannot sort, append, reverse, reorder, etc
* can only: count and index

Tuples are more efficient, they use less processing time since fewer operations are possible.

Tuples are great for "temporary variables" because they are fast and easy to work with.

Tuples can be placed on the left-hand side on an assignment statement, the parentheses can even be omitted.

(x,y) = (4, 'fred')

print y

fred

### Tuples and Dictionaries

The 'items()' method in dictionaries returns a list of tuples (key, value).

Tuples are Comparable  
The contents of a tuple can be compared and evaluated, running left to right through the listed variables. <, >, <=, >=, ==

~~Applying the sorted() function (which takes a sequence as a parameter and returns a sorted sequence) to a dictionary yields a list of its keys in sorted order; e.g.:

>>> c = {'a': 10, 'c': 22, 'b': 1, 'f': 22}

>>> sorted(c)

['a', 'b', 'c', 'f']

We can print the key:value pairs in order by key by:

>>> for k in sorted(c):

... print k, c[k]

...

a 10

b 1

c 22

f 22

We can build a list of the key:value pairs in order by key by appending each pair as a tuple to an initially empty list:

>>> csorted = []

>>> for k in sorted(c): # if c is dictionary, it CAN'T be sorted...

... csorted.append((k, c[k]))

...

>>> csorted

[('a', 10), ('b', 1), ('c', 22), ('f', 22)]

The same thing can be achieved using a tuple (with or without parentheses) as the for-loop control and building a list of tuples from c.items(), then sorting it:

>>> for k, v in c.items():

... csorted.append((k, v))

...

>>> csorted

[('a', 10), ('c', 22), ('b', 1), ('f', 22)]

>>> sorted(csorted)

[('a', 10), ('b', 1), ('c', 22), ('f', 22)]

>>> csorted = sorted(csorted)

>>> csorted

[('a', 10), ('b', 1), ('c', 22), ('f', 22)]

Since the items() method for a dictionary returns a list of its key:value pairs as tuples:

>>> c.items()

[('a', 10), ('c', 22), ('b', 1), ('f', 22)]

we could just sort c.items() instead of building a new list of tuples "by hand" first:

>>> sorted(c.items())

[('a', 10), ('b', 1), ('c', 22), ('f', 22)]

But if we want to sort the dictionary, c, by value instead of key we need to build the list of value:key pairs as tuples by hand (by reversing the value and key (v, k)), then sort that list:

>>> tmp = list()

>>> for k, v in c.items():

... tmp.append( (v, k) )

...

>>> print tmp

[(10, 'a'), (22, 'c'), (1, 'b'), (22, 'f')]

>>> tmp.sort(reverse=True)

>>> print tmp

[(22, 'f'), (22, 'c'), (10, 'a'), (1, 'b')]

When sorting by value and there are duplicate values, the second item in the tuple defines the order of the duplicates.

c = {'a':10, 'b':1, 'c':22, 'f':22}

tmp = list()

for k, v in c.items() :

tmp.append( (v, k) )

print tmp

#[(10, 'a'), (22, 'c'), (1, 'b'), (22, 'f')]

tmp.sort(reverse=True)

print tmp

#[(22, 'f'), (22, 'c'), (10, 'a'), (1, 'b')]

#because 'f' comes before 'c' when sorting in reverse order.

## List Comprehension

List comprehension (represented by [,]) creates a dynamic list.

print sorted( [ (v,k) for k,v in c.items() ] )

# Chapter 11

<https://docs.python.org/2/howto/regex.html>

# Chapter 12

Get Web data

import socket

mysock = socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)

mysock.connect((‘www.py4inf.com, 80’))

mysock.send(‘GET <http://www.py4inf.com/code/romeo.txt> HTTP/1.0\n\n’)

while True:

data = mysock.recv(512)

if ( len(data)<1) :

break

print data

mysock.close()

reading web page

import urllib

fhand = urllib.urlopen(‘http://www.py4inf.com/code/romeo.txt’)

for line in fhand:

print line.strip()

$ telnet [www.dr-chuck.com](http://www.dr-chuck.com) 80

GET <http://www.dr-chuck.com/page1.htm> HTTP/1.0

Web Scraping (internet parsing)

Use Beautifulsoup to retrieve data from html

<http://www.crummy.com/software/BeautifulSoup/>

import urllib

from BeautifulSoup import \*

url = raw\_input(‘Enter –‘)

# read all, html is a string

html = urllib.urlopen(ur).read()

soup = BeautifulSoup(html) # give back a soup object

# retrieve a list of the anchor tags – e.g. <a href =”....”>, </a> etc.

# each tag is like a dictionary of HTML attributes

tags = soup(‘a’)

for tag in tags

print tag.get(‘href’, None)

This course uses Python 2.x and the version of Beautiful Soup supported for this course is BeautifulSoup 3.

The sample code for this course that you find at <http://www.pythonlearn.com/code/> (urllinks.py, etc) and the files provided for this week's assignments are written in Python 2.x and use BeautifulSoup version 3. To use this sample code, you should download Beautiful Soup from

<http://www.pythonlearn.com/code/BeautifulSoup.py>

and place it in the same folder as your programs that use the command:



1

from BeautifulSoup import \*

No other installation steps are required.

BeautifulSoup 3 is specifically written for Python 2.x and will not work in Python 3.x. Students who choose to use Python 3.x for the assignments will have to download and figure out how to install BeautifulSoup 4.

You can download BeautifulSoup 4 from <http://www.crummy.com/software/BeautifulSoup/> and follow the instructions there for installation. You will also need to adapt our sample code to make it work with the Python 3.0 BeautifulSoup library. The syntax will be different and that syntax is not supported by the course materials. Please note that the staff will not be able to help you with any issues you might encounter using either Python 3.x or BeautifulSoup 4.

# Chapter 13

XML

Tags – indicate the beginning and end of element

Attributes – Keyword/value pairs on the opening tag of XML

Serialize / De-Serialize – Convert data in one program into a common format that can be stored and /or transit between systems in a programing language - independent manner

Exercise 13-1:

**Extracting Data from XML**

In this assignment you will write a Python program somewhat similar to <http://www.pythonlearn.com/code/geoxml.py>. The program will prompt for a URL, read the XML data from that URL using **urllib** and then parse and extract the comment counts from the XML data, compute the sum of the numbers in the file.

We provide two files for this assignment. One is a sample file where we give you the sum for your testing and the other is the actual data you need to process for the assignment.

* Sample data: <http://python-data.dr-chuck.net/comments_42.xml> (Sum=2553)
* Actual data: <http://python-data.dr-chuck.net/comments_338796.xml> (Sum ends with 82)

You do not need to save these files to your folder since your program will read the data directly from the URL. **Note:** Each student will have a distinct data url for the assignment - so only use your own data url for analysis.

**Data Format and Approach**

The data consists of a number of names and comment counts in XML as follows:

<comment>

<name>Matthias</name>

<count>97</count>

</comment>

You are to look through all the <comment> tags and find the <count> values sum the numbers. The closest sample code that shows how to parse XML is [geoxml.py](http://www.pythonlearn.com/code/geoxml.py). But since the nesting of the elements in our data is different than the data we are parsing in that sample code you will have to make real changes to the code.

To make the code a little simpler, you can use an XPath selector string to look through the entire tree of XML for any tag named 'count' with the following line of code:

counts = tree.findall('.//count')

Take a look at the Python ElementTree documentation and look for the supported XPath syntax for details. You could also work from the top of the XML down to the comments node and then loop through the child nodes of the comments node.

**Sample Execution**

$ python solution.py

Enter location: http://python-data.dr-chuck.net/comments\_42.xml

Retrieving http://python-data.dr-chuck.net/comments\_42.xml

Retrieved 4204 characters

Count: 50

Sum: 2...

JSON

import json

data = ‘’’{

...

}’’’

info = json.loads(data) # load data into dictionary

print info[“name”]

print infor[“email”][“hide”]

JSON is more simple but XML is more expressive in complicated structures.

API application program interface – the fine set if rules to interface with an application program

SOAP – simple object access protocol (old and crappy)

REST – Representation State Transfer (resource focused)

<http://maps.googleapis.com/maps/api/geocode/json?sensor=false&address=Ann+Arbor%2C+MI>

+ means space, %2C means comma ,

Object Oriented

Class, Method, instance

Class, constructor (in the beginning) is common. Destructor (in the end) is rare.

When defining function in class, “self” is aliasing to the instance when calling the function in code.

Inheritance:

Child class inherits all properties of mother class.

SQLite exercise 1

# Instructions

If you don't already have it, install the SQLite Browser from <http://sqlitebrowser.org/>.

Then, create a SQLITE database or use an existing database and create a table in the database called "Ages":

CREATE TABLE Ages (

name VARCHAR(128),

age INTEGER

)

Then make sure the table is empty by deleting any rows that you previously inserted, and insert these rows and only these rows with the following commands:

DELETE FROM Ages;

INSERT INTO Ages (name, age) VALUES ('Aadam', 17);

INSERT INTO Ages (name, age) VALUES ('Arann', 28);

INSERT INTO Ages (name, age) VALUES ('Ayomide', 29);

INSERT INTO Ages (name, age) VALUES ('Komal', 39);

INSERT INTO Ages (name, age) VALUES ('Heddle', 15);

Once the inserts are done, run the following SQL command:

SELECT hex(name || age) AS X FROM Ages ORDER BY X

Find the **first** row in the resulting record set and enter the long string that looks like **53656C696E613333**.

**Note:** This assignment must be done using SQLite - in particular, the SELECT query above will not work in any other database. So you cannot use MySQL or Oracle for this assignment.

Exercise:

# Musical Track Database

This application will read an iTunes export file in XML and produce a properly normalized database with this structure:

CREATE TABLE Artist (

id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,

name TEXT UNIQUE

);

CREATE TABLE Genre (

id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,

name TEXT UNIQUE

);

CREATE TABLE Album (

id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,

artist\_id INTEGER,

title TEXT UNIQUE

);

CREATE TABLE Track (

id INTEGER NOT NULL PRIMARY KEY

AUTOINCREMENT UNIQUE,

title TEXT UNIQUE,

album\_id INTEGER,

genre\_id INTEGER,

len INTEGER, rating INTEGER, count INTEGER

);

If you run the program multiple times in testing or with different files, make sure to empty out the data before each run.

You can use this code as a starting point for your application: <http://www.pythonlearn.com/code/tracks.zip>. The ZIP file contains the **Library.xml** file to be used for this assignment. You can export your own tracks from iTunes and create a database, but for the database that you turn in for this assignment, only use the **Library.xml** data that is provided.

To grade this assignment, the program will run a query like this on your uploaded database and look for the data it expects to see:

SELECT Track.title, Artist.name, Album.title, Genre.name

FROM Track JOIN Genre JOIN Album JOIN Artist

ON Track.genre\_id = Genre.ID and Track.album\_id = Album.id

AND Album.artist\_id = Artist.id

ORDER BY Artist.name LIMIT 3

The expected result of this query on your database is:

|  |  |  |  |
| --- | --- | --- | --- |
| **Track** | **Artist** | **Album** | **Genre** |
| Chase the Ace | AC/DC | Who Made Who | Rock |
|  |  |  |  |
| D.T. | AC/DC | Who Made Who | Rock |
|  |  |  |  |
| For Those About To Rock (We Salute You) | AC/DC | Who Made Who | Rock |
|  |  |  |  |