Python Summary Sheet #1

(LP =Learning Python, 5th Ed, by Lutz)

Help in Python (LP 443)

dir(*m*)	quick list of available attributes [x for x in dir(*obj*) if not x.startswith('__')]
help(*m*)	man page for module *m* or help(*mod.attr*)
help(*f*)	man page for function *f*

Linux Executable Scripts (LP 59)

Normal Python scripts with...

1. First line: **#!/path/to/python** *(Python comment)*
2. Must be executable (chmod 755 *file.py*)

Operator Precedence Table

func_name(*args*, ...)	Function call (LP 530ff)
x.attribute	Attribute reference
**	Exponentiation
*, /, %	Mult, divide, modulus
x / y	(2.X) classic (truncates) (3.X) true (rem. w/ float)
x // y	floor (always trunc. *down*)
+, -	Add, subtract
>, <, <=, >=, !=, ==	Comparison
in, not in	Membership tests
not, and, or	Bool. NOT, AND, OR
+=	Augment (LP 350-352)

Module Import (LP 688-702)

import *module_name*
from *module_name* import *name1*, *name2*, ...
from *module_name* import *
reload(*module*) for 2.X | imp.reload(*module*) for 3.X

Prevent module execution on import: (LP 749)

if __name__== "__main__": *run whatever stmts*

If the file is run as a top-level program, __name__ is set to __main__ when it starts. ***Use for self-test code.***

If the file is imported as a module, __name__ is set to the module's name as know by its clients.

Common Data Types

Type	Description	Literal Ex
int	32-bit Integer	3, -4
long	Integer > 32 bits	101L
float	Floating point number	3.0, -6.55
complex	Complex number	1.2J
bool	Boolean	True, False
str	Immutable char sequence	'Python'
tuple	Immutable ordered seq.	(2, 4, 7)
list	Mutable, ordered sequence	[2, x, 3.1]
dict	Mapping by *key*:*value*	{x:2, y:5}

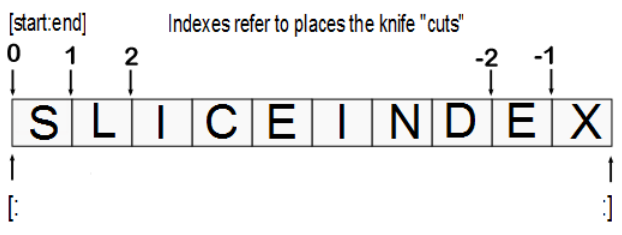
Literal [def]: (LP 95) an expression whose syntax generates an object—sometimes called a *constant*.

Print in Python 3.X (LP 359)

print(obj*, sep=' ', end='\n', file=sys.stdout)

obj	object(s) to be printed (*comma separated)
sep	string inserted between each object's text
end	string added to the end of the printed text
file	specifies the file to which the text is sent
ex:	print(x, y, z, sep=', ') print(x, y, z, sep=', ', end='...\n') Print to file: print(x, y, z, file=open('data.txt', 'w') Dispaly file text: print(open('data.txt').read())

Slicing (LP 202)



x[i]	fetch item at offset (indexing)
x[i:j]	extract contiguous section
x[i:j:k]	k = step/stride (default: +1)
[:0]	create empty slice, same type

Common Syntax Structures (LP 320)

Assignment Statements

var = *exp* (assign *exp* to *var* variable)
L[*index*] = *value* (assign *value* to *index* in list L)
D[*key*] = *value* (assign *value* to *key* in dict D)

Console Input ↓(3.X) ↓(2.X)
var = input([*prompt*]) or *var* = raw_input([*prompt*])

Iteration Keywords (LP 389)

pass -	placeholder for future stmts
continue -	stop current iteration, return to top
break -	jumps out of enclosing loop

Selection (LP 372ff)

```
if (boolean_exp):   or   if not(boolean_exp):
    stmt ...
[elif (boolean_exp):
    stmt ...] ...
[else:
    stmt ...]
```

Repetition (LP 387ff)

```
while (boolean_exp):
    stmt ...
```

Traversal (LP 395)

```
for var in traversable_object:
    stmt ...
```

Function Definition (LP 530ff)

```
def function_name( parmameters ):
    stmt ...
```

Function Call (LP 530ff)

```
function_name( arguments )
```

Class Definition

```
class Class_name [ (super_class) ]:
    [ class variables ]
    def method_name( self,  parameters ):
        stmt ...
```

Object Instantiation

```
obj_ref = Class_name( arguments )
```

Method Invocation (dot notation)

```
obj_ref.method_name( arguments )
```

Exception Handling

```
try:
    stmt ...
except [exception_type] [, var]:
    stmt ...
```

Common Built-in Functions

Function	Returns
abs(*x*)	Absolute value of *x*
dict()	Empty dictionary, **ex:** d = dict()
chr(#)	character of Unicode #
enumerate(*iter*)	results in tuple *(count, item)*
filter(*func*, *iter*)	Bool. Apply *func* to each *i* in *iter*
float(*x*)	Convert int or string *x* to float
id(*obj*)	memory address of *obj*
int(*x*)	Convert float or string *x* to int
len(*s*)	Number of items in sequence *s*
list()	Empty list, **ex:** L = list()
map(*func*, *iter*)	apply *func* to each item in *iter*
max(*s*)	Maximum value of items in *s* (seq)
min(*s*)	Minimum value of items in *s* (seq)
open(*f*, '*mode*')	Open filename *f*. Common modes: 'r' = open for reading (default) 'w' = open for writing (truncate 1st) 'a' = open for writing (append) '+' = open for update (read+write)
ord(*char*)	ASCII code of *char*
pow(*x*,*y*)	*x* ** *y* (to the power of, exponent)

range(*x*)
 	
range(*x, y, z*)
	In 3.X, returns an iterable of *x* ints, 0 to *x*-1 (defaults). 2.X: returns list
	x=start val, y=stop val, z=step val

reversed(*seq*)	Returns reverse seq of an iterable
round(*x*,*n*)	floating pt. *x* rounded to *n* places
sorted(*iter*)	New list of sorted items. Reverse: sorted(*iter*, reverse=True)
str(*obj*)	str representation of *obj*
sum(*seq*)	Sum of numeric sequence *seq*
tuple(*items*)	tuple of *items*
type(*obj*)	Data type of *obj*
zip(*iter*, *iter*)	tuple of successive items in *iter*

Lambdas (LP 567)

Defines functions *in-line* w/o a def statement.

Syntax:
lambda para1, para2, … paraN : exp using paras

Example:
f = (lambda x, y, z : x + y + z) | f(2,3,4) results in 9

Example:
f = (lambda x: x ** 2) | f(4) results in 16

DocStrings (LP 446ff)

Coded as strings at the ***top*** of module files, function statements, and class statements. The help() func. displays DocStrings like a man page.

top of module file	""" Program: name.py Author: FirstName LastName Last Date Modified: mm/dd/yyyy Give a purpose statement, then: 1. CONSTANTS 2. Inputs 3. Computations 4. Outputs """
top of functions	def func(*parameters*): """ Put multiline functions in triple quotes to explain func. """
top of classes	class Whatever: "Single lines in quotes"
# comments	Use hashmark comments for smaller scale documentation. For example: 1. Precede major segments of code w/ purpose statement 2. Line comments to explain variables, ambiguous code, etc.

LIST COMPREHENSIONS

Whenever you think about performing an operation on each item in a sequence, think ***comprehension***.

Basic Syntax (a "backward for-loop") (LP 425, 584)

syntax: [*expression* for *target* in *iterable*]

example: L = [x + 10 for x in *iterable*]

[ ]	Square brackets, b/c it constructs a new list
x + 10	Beg. w/ an arb. exp. using a loop var (x)
for	begins a for loop w/ var (x) and iterable (L)
iter	Any iterable object: string, list, dict, tuple...
L =	Results in a new list

Examples (using files)

Clean up lines of a file using rstrip():

```
f = open('script2.py')
lines = f.readlines()           # Return list of all lines
lines = [ line.rstrip() for line in lines ]
```

Or... all of the above in one comprehension:

```
lines = [ line.rstrip() for line in open('script2.py') ]
```

Or... same, but with "method chaining":

```
[ line.rstrip().upper() for line in open('script2.py') ]
```

Extended Syntax: Filter Clauses (if) (LP 427, 584)

Use if-clause as a filter within the for-loop.

syntax: [*exp* for *target* in *iterable* if *condition*]

Example: Collect only lines that begin with letter 'p'

[line.rstrip() for line in open('f.txt') **if line[0] == 'p'**]

If the line begins with 'p' it is passed to the rstrip()

Example: Give total number of lines in a text file

```
f = r'd:\books\draft.txt'
len( open(f).readlines() )
```
=> *Will give results including blank line*

```
len( [ line for line in open(f) if line.strip() != ' ' ]
```
=> *Will give results excluding blank lines*

Dictionary Comprehension (LP 265)

```
D = { x: x*2  for  x  in  range(10) }
```

Python Summary Sheet #2

(LP =Learning Python, 5th Ed, by Lutz)

Common String Methods

S.method()	**Returns (str unless noted)**
S.capitalize()	*S* with first char uppercase
S.center(*w*)	*S* centered in str *w* chars wide
S.count(*sub*)	int number of non-overlapping occurrences of *sub* in *S*
S.find(*sub*)	int index of first occurrence of *sub* in S or -1 if not found
S.isdigit()	Boolean True if *S* is all digit chars, False otherwise

S.islower()
S.isupper()
	Boolean True if *S* is all lower/upper case chars, False otherwise

S.join(*seq*)	All items in *seq* concatenated into a str, delimited by S (**ex:** ' '.join(seq))

S.lower()
S.upper()
	Lower/upper case copy of *S*

S.strip()
S.lstrip()
S.rstrip()
	Copy of *S* with leading (left) and/or trailing (right) white space removed

S.replace(x,y)	Replace 'x' with 'y' (put in quotes)
S.split([*sep*])	List of tokens in *S*, delimited by *sep*; if no *sep* given, split on white space
S.startswith()	Returns True if startswith *string*
S.endswith()	Returns True if endswith *string*

Raw Strings (LP 196-198)

Need b/c a backslash (\ ) denotes a spec char (LP 195). Raw strings are used to turn off the escape mechanism.

syntax	**r** or **R** in front of '*string*'
Windows Paths	*var* = open(r'C:\filename.txt')
Regular Expressions	*var* = re.findall(r'\b')

String Formatting Expressions (LP 216)

Expression Syntax
'…%s...' % (tuple, of, values)

%s
Every object type works with the string code. Unless you need special formatting for numbers, use %s for everything.

Example:
'That is %d %s bird!' % (1, 'dead') or
'That is %s %s bird!' % (1, 'dead')
both return: **That is 1 dead bird!**

Common String Formatting TypeCodes (LP 219)

%**s**	String (or any object's str(x) string)
%**d**	Decimal (base-10 number)
%**i**	Integer
%**f**	floating-point decimal

String Formatting Left-Side Syntax (LP 219-220)

%[(*keyname*)][*flags*][*width*][.*precision*]*typecode*

keyname	key name for indexing the dictionary given on the right side of the expression (optional)
flags	specify (optional): - = left justification + = numeric sign (either “-” or “+”) 0 = pad space with zeros for numbers = blank left before positive number
width	total min. field width for text (optional)
.precision	# of digits to display after decimal (opt)
typecode	see below (s, d, i, f) - ***required***

String Formatting Syntax Examples

x = 123, y = 456.789

“%6d” % x	...123	(width: 6, decimal #)
“%06d” % x	000123	(width: 6, pad w/0's)
“%8.2f” % y	..456.78	(width: 8, 2 dec. pts.)
“%-8s” % “Hello”	Hello...	(left just., width: 8)

Common List Literals & Operations (LP 240)

Operation	**Interpretation**
L = [ ] or L = list()	Create an empty list
L = list('spam')	List of iterable's items
L = list(range(5))	List of successive integers
L[i]	Index
L[i] [j]	Index of index (nested lists)
L[i : j]	Slice
len(L)	Length
L1 + L2	Concatenate
L * 3	Repeat
for x in L: print(x)	Iteration
3 in L	Boolean membership test.
del L[i] or L[i:j]	Delete items from list L
L[i] = *obj* L[1:2] = [4,5] L[1:1] = [6,7] L[1:2] = [ ]	Assign *obj* to L at index *i* (LP 244) Replacement / Insertion(LP 245) Insertion (replace nothing) Deletion (insert nothing)

Common List Methods

L.method()	**Result/Returns**
L.append(*obj*)	Append ***single item*** to end of L
L.count(*obj*)	Search: Returns int number of occurrences of *obj* in *L*
L.copy()	Return shallow copy of list L
L.extend(*list*)	Append ***mult. items*** to end of L
L.index(*obj*)	Search: Returns index of first occurrence of *obj* in *L*;
L.insert(*i*, *X*)	Insert *X* at index *i*.
L.pop([*index*])	Returns item at specified *index* or item at end of *L* if *index* not given; raises IndexError if *L* is empty or *index* is out of range
L.remove(*obj*)	Removes 1st occurrence of *obj* from *L*
L.reverse()	Reverses *L* in place
L.sort()	Sorts *L* in place (see **sorted()**)

Common Dictionary Literals & Operations (LP 252)

Operation	Interpretation
D = {} or D=dict()	Create an empty dictionary
D={'name':'Bob'}	Create a one-item dictionary
D=dict(age=30)	Create a one-item dictionary
D['name']	Index by key (pulls out value)
D['name']['other']	Index of index (nested dict.)
'age' in D	Bool. membership test **by key**
len(D)	Return number of k:v pairs in D
D[*key*] = *val*	Set D[*key*] to *val* (add/change)

Common Dictionary Methods

D.method()	**Result/Returns**
D.clear()	Remove all items from *D*
D.copy()	copy(top-level)
D.get(*k* [,*val*])	Ret. *D*[*k*] if *k* in *D*, else *val* default ***histogram***: d[c] = d.get[c,0] +1
D.has_key(*k*)	Return True if *k* in *D*, else False
D.items()	Return <u>*view object*</u> of key-val pairs in *D*; each list item is 2-item tuple. 3.X force list => list(D.items()) **ex:** for (k,v) in list(d.items()): *stmts*
D.keys()	Return <u>*view object*</u> of *D*'s keys 3.X force list => list(D.keys())
D.pop(*k*, [*val*])	Remove & return key *k*, return mapped value or *val* if *k* not in *D*
D.popitem()	Remove & ret. an arbitrary k,v pair
D.update(D2)	merge by keys (overwrite existing)
D.values()	Return <u>*view object*</u> of *D*'s values 3.X force list => list(D.values())

String Backslash Characters (LP 195)

\*newline*	Ignored (continuation line)		
\ooo	character with octal value ooo		
\xhh...	character with hex value hh...		
\\	Backslash (\\)	**\f**	Formfeed (FF)
\\'	Single quote (')	**\n**	Linefeed (LF)
\\"	Double quote (")	**\r**	Carriage Return (CR)
\a	Bell (BEL)	**\t**	Horizontal Tab (TAB)
\b	Backspace (BS)	**\v**	Vertical Tab (VT)

Common Tuple Literals & Operations (LP 276)

Operation	Interpretation
T = (0,)	One-item tuple
T = (0, 'Ni', 1.2, 3)	Four-item tuple
T = 0, 'Ni', 1.2, 3	Four-item tuple (same as above)
T = tuple('spam')	Creates tuple: ('s', 'p', 'a', 'm')
T[i]	Index
T[i] [j]	Index of index
T[i : j]	Slice
len(T)	Length
T1 + T2	Concatenate
T * 3	Repeat
for x in T: print(x)	Interation
'spam' in T	Boolean membership test

Common Tuple Methods

T.method()	**Returns**
T.count(*obj*)	Returns # of occurrences of *obj* in *T*
T.index(*obj*)	Returns index of first occurrence of *obj* in *T*; ValueError if *obj* is not in *T*

Regular Expressions

^ Anchors to beginning of search str.
$ Anchors to end of search str.
. Matches any character (a wildcard).
***** Repeats preceding character 0 or more times (greedy).

Compile for complex expressions:
r = re.compile(*regex*)

***?** Repeats preceding character 0 or more times (non-greedy)
+ Repeats preceding character 1 or more times (greedy).
+? Repeats preceding character 1 or more times (non-greedy)
[aeiou] Matches a single character in the specified set.
[a-z0-9] Matches a single character in the specified range.
[^A-Za-z] Matches a single character NOT in the set.
(Indicates where the string extraction is to start.
) Indicates where the string extraction is to end.
\d Match any decimal digit: **[0-9]**. **\D** non-digit char:**[^0-9]**.
\w Match any word (alphanum) char. **\W** Match non-word char.
\s Match any whitespace char. **\S** Match non-whitespace char.
\b Matches word (alphanum sequence) boundary.
\B Matches non-word boundary (every position \b does not).

re.search(*regex, string*) Scan a string (returns True or False).
re.findall(*regex, string*) Extract data from a str. Returns a list.

Common File Methods (LP 283)

F.method()	**Result/Returns**
F.read([*n*])	Return str of next *n* chars from *F*, or up to EOF if *n* not given
F.readline([*n*])	Return str up to next newline, or at most *n* chars if specified
F.readlines()	Return list of all lines in *F*, where each item is a line
F.write(*s*)	Write str *s* to *F*
F.writelines(*L*)	Write all str in seq *L* to *F*
F.close()	Closes the file
open(*f*, '*mode*')	Open filename *f*. Common modes: 'r' = open for reading (default) 'w' = open for writing (truncate 1st) 'a' = open for writing (append) '+' = open for update (read+write)

File Specifics (LP 282-286)

File objects serve as <u>*links*</u> to files on hdd.
File objects allow transfer of <u>*strings*</u> only.
File objects are <u>*iterable*</u> by line.

File Example

```
myfile = open('file.txt', 'w')          #create link to file
myfile.write('hello text file\n')       #write string to file
myfile.write('goodbye text file\n')     #write string to file
myfile.close()                          #flush buffers
```

pickle module (LP 290)

Store any object in a file w/o converting to string.

Example: pickle a dictionary...

```
import pickle
D = {'a' : 1, 'b' : 2}
F = open('datafile.pkl', 'wb')    # wb = write, binary
pickle.dump(D, F)           #pickle object D to file F
F.close()
```

Example: un-pickle the dictionary...

```
F = open('datafile.pkl', 'rb')      # rb = read, binary
E = pickle.load(F)             # load pickled object F
print(E) >> {'a' : 1, 'b' : 2}
```