BIG DADDY'S PYTHON TOOLBOX For 3.5

Reserve Words

Comparsion / Conjunction
True, == (equal), **False, none** (i.e., null), **and, not, or,** **in** list, tuple, string, dictionary **is** True if **same** object, **is not**

Definition
class create a class
def create a function
del items in lists (del mylist[2]), whole strings, whole tuples, whole dictionaries

Module Management
import connects module, ex: import math
from gets a function from math import cos
as creates an alias for a function

Miscellaneous
pass (placeholder – no action)
with wrapper ensures **_exit_** method

Functions
def, return(obj), yield, next
def creates; inside functions **yield** is like **return** but returns a generator whose sequential results are triggered by **next**;
global declares global var in a function
non local a variable inside a nested function is good in the outer function
lambda anonymous inline function with no return statement

```
a = lambda x: x*2
for z in range (1,6):
    print (a (z) )
```

Error Management
raise forces a ZeroDivisionError
try except else finally assert
used in error handling blocks
try: code with error potential
except: do this if you get the error
else: otherwise do this code
finally: do this either way
assert: condition=False raises **AssertionError**

Looping
while (some statement is true)
for alist=['Be','my','love']
```
    for wordnum in range(0,len(alist)):
        print(wordnum, alist[wordnum]) #slice
```
range (start, stop, [step])
See data container functions
break ends the smallest loop it is in;
continue ends current loop iteration

Decision Making
if elif else
```
def if_example(a):
    if a == 1:
        print('One')
    elif a == 2:
        print('Two')
    else:
        print('Some other')
```

The Ternary if Statement
An inline **if** that works in formulas:
myval = (high if (high > low) else low) * 3

Multi-line Statements \
Not needed within [], {}, or ()
Multiple Statements on a Line ; not with statements starting blocks like **if**

CLASS: Your own data container. DEFINE DESIGN:
class Name (inheritance **object**)
  def __init__(self, mandatory variables,…)
    accessname = **mandatory variable** …repeat as necessary
  Other functions:"get**accessname**(self)", or "return self.varible"
CREATE INSTANCE:
MyInstanceName = ClassName(mandatory variables values)
ACCESS INSTANCE DATA:
Print(MyInstanceName.accessname in get funtion)

Major Built-In Functions

String Handling (🖐=converts/returns)
str(object) 🖐 string value of object
repr(object) 🖐 printable representation string
ascii(str) 🖐 like repr but escape non-ascii
eval(expresion) 🖐 value after evaluation
chr(i) 🖐 character of Unicode [chr(97) = 'a']
input(prompt) 🖐 user input as a string
len(—) 🖐 length of str, items in list/dict/tuple
ord(str) 🖐 value of Unicode character
slice -> Xx[start: stop [:step]] 🖐 a new object selected by slice selection, Xx= "Python"; Xx[2:5] 🖐 tho; Xx[:2] 🖐 py; Xx[2:] 🖐 thon; Xx[::2] 🖐 pto
format(value [,format_spec]) 🖐 value in a formatted string—**extensive and complex** -
'{:,}'.format(1234567890) yields '1,234,567,890'
'{:.3%}'.format(11.23456789) yields '1123.457%'
'{:*^50}'.format("right aligned") {:}-format string follows, * - fill character, ^ - alignment (^=centered), 50 - width
Also1: substitution: 'A couple: {him} and {her}'.format (him='Bo',her='Jo') Also2: number format: b | c | d | e | E | f | F | g | G | n | o | s | x | X | %

String Format Operator: %
Depricated: use str.format() above, however:
% is used with print to build formatted strings
print ("My horse %s has starting slot %d!" % ('Arrow', 5))
Where the % character can format as: **%c** char-acter, **%s** string, **%i** signed integer decimal, **%d** signed integer decimal, **%e** exponential notation, **%E** exponential notation (upper cs), **%f** floating point real number, **%g** the shorter of %f and %e, **%G** the shorter of %F and %E
also: ***** specifies min field width, **-** left justifica-tion, **+** show sign

Number Handling
abs(x) 🖐 absolute value of x
bin(x) 🖐 integer to binary bin(5)= '0b101'
(one 4, no 2's, one 1) bin(7)[2:]= '111'
divmod(x,y) takes two (non complex) numbers as arguments, 🖐a pair of numbers - quotient and remainder using integer division.
float(x) 🖐 a floating point number from a number or string
hex(x) 🖐integer to hex string hex(65536)=0x10000 or hex(x)[2:]='10000' also **oct(x)** 🖐int to octal
int(x) 🖐 an integer from a number or string
pow(x,y [,z]) 🖐 x to y, if z is present returns x to y, modulo z pow(2,7)=128, pow(2,7,3)=2
round(number [,digits]) 🖐 floating point number rounded to digits; Without digits it re-turns the nearest integer. Round(3.14159,4)=3.1416

Miscellaneous Functions
bool(x) 🖐 true/false, 🖐 false if x is omitted
callable(object) 🖐true if object is callable
help(object) invokes built-in help system, (for interactive use)
id(object) 🖐unique object integer identifier
print(*objects, sep=' , end='\n', file= sys.stdout, flush=False) prints objects separated by sep, followed by end;

File open (and methods)
fileobject=**open**(file [,mode],buffering]) The basic modes: **r, r+, w, w+, a** …more
file object methods: **.read(size), .readline, .readlines, list(fo), .write(string), .close, .splitlines**
with open("C:\Python351\Jack.txt",'r+') as sprattfile:
    sprattlist=sprattfile.read().splitlines() *<- removes '/n'*
print(sprattlist)
🖐['Jack Spratt', 'could eat ', 'no fat.', 'His Wife', 'could eat ', 'no lean.'] *The WITH structure auto closes the file.*

Operators

Math: **+, -, *, /, //** (floor or truncated division), ****** (expo-nent), **%** (mod or modulo returns the remainder) **x = 8%3; print(x)** 🖐 2
Boolean/Logical: **and, or, not**
Comparison:**<, <=, >, >=, is, is not, ==** (same), **!=**(not equal)
Sequence Variable Opers: **in** (true if found, **not in, +** concatenation, ***** repetition, s**[i]** slice, s**[i:j]** range slice, **r** (r'str' – raw string suppresses ESC chars)
Identity: **is/is not** checks for same object
Bitwise: **&, |** (or), **^** (xor), **~** bits inverted, **<<** (shift left n bits), **>>**(shift right n bits)
Assignment:(execute & assign)
-=, +=, *=, /=, **=, %=, //=, =, (only **+** & **-** work for strings)

Other Functions
vars(), dir(), super(), globals(), setattr(), bytearray(), classmethod(), zip(), locals(), __import__(), object(), memoryview(), hasattr(), issubclass(), isinstance(), compile(), hash(), complex(), bytes(), exec(), frozenset(), delattr(), property(), getattr(), staticmethod()

String Methods
.find(sub[, start[, end]])
🖐First char BEFORE sub is found or -1 if not found ex: aword = "python"; print (aword.find("th")) 🖐 2
.capitalize() 🖐first character cap'ed
.lower() 🖐 a copy of the string with all text converted to lowercase.
.center(width[, fillchar]) string is centered in an area given by width using fill character 'fillchar'
.ljust(width [, fillchar]) or **.rjust()**
.count(sub[, start[, end]]) number of substrings in a string
.isalnum() .isnumeric() .isalpha .isdigit() .isspace() .islower() .isupper .isprintable() may be null
🖐 true if all char meet condition and variable is at least one char in length
.replace(old, new[, count]) 🖐 a copy of the string with substring old replaced by new. If opt argument count is given, only first count are replaced.
.rfind(sub[, start[, end]]) 🖐 the highest index in the string where substring sub is found, contained within slice [start:end]. Return -1 on failure.
.strip([chars]) 🖐 a copy of the string with the leading and trailing characters removed. The chars argument is a string specifying the set of characters to be removed. If omitted or None, the chars argument removes whitespace.
.zfill(width) 🖐 a copy of the string left filled with ASCII '0' digits to make a string of length width. A leading sign prefix ('+'/'-') is handled by inserting the padding after the sign character rather than before. The original string is returned if width is less than or equal to len(str).
str.split() - separates words by space

Data Containers
Methods / Operations

Below: (i/j/k-> index; x->item or object; L/T/D/S->name of list, tuple, dictionary, or set.

LISTS: create – **[x,x,...]**; **.insert**(i,x); **append**(x); **L[i]=x**; **.extend**(x,x,…); **.remove**(x); **del L**; **.pop()**; **.pop**(i); L[i] =*replacement* x; L[i:j]=[x,x…] *replace multi-items*; i=**L.index(x**[,*at or after index* i [,*before index* j]]) *retrieve index number of first value of x*; V=**iter**(L) *creates iteration generator*; **next**(V,default) *to step thru iteration*; **len**(L); **.count**(x); **.max**(L), **min** (L); if v **in** L *determine membership*; **.copy()**; **sort**(key=none, reverse=False); **.reverse**; **.clear**; L=[]; **del L**; L=**list**(tuple)

TUPLES: create – **(x,[x],(x),...)** *objects can include lists and other tuples*;**+=** *add items*; **+=(x,)** *add singe item*; **tuple[i:j]** *start is 0, end j-1*; **x,x,…=T[i:j]** *retrieve values*; i=**T.index(x**[,*at or after index* i [,*before index* j]]); **for int in T**; v=**iter(T)** *creates iteration generator*; **next**(v) *next iteration*; **len**(T); **.count**(x); **.max**(T); **.min**(T); x **in** T; **sorted**(T, **reverse=False**); T[::-1]; T=() *clears all values*; **del** T; T=**tuple**(somelist) *creates a tuple from a list*

DICTIONARIES: create - {k:v, k:v,…} ; D=**dict.fromkeys**(keys/list[,values]); D.**update** (D2) *adds D2 to D*; **D[k]**=v *returns value of k*; **del D[k]** *deletes key and item*; D.**pop**(k [,default]); D.**popitem()**; D.**items()**; D.**keys()**; D.**values()**; D.**get**[k] *same as D[k]*; v=**iter**(D) *creates iteration variable*; **next**(v) *step thru iterations*; **len**(D); v **in** D; v not in D; D.**has_key**(v); D.**copy()**; D.**clear()**; **del** D; D.**setdefault(k[,default])** *if k is already in the dictionary return the key value, if not, insert it with default value and return default*

SETS: create – S=**set**(x,x,…) *no duplicate items*; S=**set**(L) *take list as set items*; S=**"some text string**" *yields unique letters*; S=**set()**; S.**union**(S2); S.**update**(S2); S.**intersection**(S2); S.**difference**(S2); S.**add** (x); S.**remove**(x) *gives KeyError is not present*; S.**discard**(x); S.**pop()**; S.**isdisjoint** (S2) *true if no common items*; S.**issubset**(S2) *or* **S<=S2** *contained by*; **S<S2** *true if both S<=S2 and S!=S2 (is not equal)*; S.**issuperset** (S2) *or* **S>=S2**; **S>S2**; v=**iter**(S) *create iteration variable*; next(v); **len**(S); S **in**; S **not in**; S.**copy()**; S.**clear()**; **del** S

Escape Characters

Nonprintable characters represented with backslash notation: r ignores esc chars; print(r'test1\t\n test2') ⇨ test1\t\n test2
\a bell or alert, **\b** Backspace, **\s** Space, **\cx** or **\C-x** Control-x, **\e** Escape, **\M-\C-x** Meta-Control-x, **\f** Formfeed, **\n** Newline, **\t** Tab, **\v** Vertical tab, **\x** Character x, **\r** Carriage return, **\nnn** Octal notation, where range of n is 0-7 **\xnn** Hexadecimal notation, n is in the range 0-9, a-f, or A-F

Basic Programming Examples:
www.wikipython.com

Data Container Functions

all(iterable) ⇨ TRUE if all elements are true
any(iterable) ⇨ TRUE if any element is true
both all and any are FALSE if empty
enumerate(iterable, start = 0)⇨list
alst = ['x','y','z']
print(**alst**(enumerate(**blst**)))
⇨ **[(0,'x'), (1,'y'), (2,'z')]**

type([iterable]) ⇨a datatype of any object
max(type) min(type) - **not** for ~~tuples~~
sum(iterable [, start]) must be all numeric, if a=[8,7,9] then sum(a) returns 24
sorted(iterable [,key=][,reversed])
reversed is Boolean with default False; strings without **key** sorted alphabetically, numbers high to low; key examples: print (sorted(strs, key=len)) sorts by length of each str value; ex: key= strs.lower, or key = lambda tupsort: tupitem[1]
reverse() reverses in place; mylist.**reverse()**
reversed() reverses access order—list or tuple

```
alist=["Amy","Bo","Cy"]          Cy
alist.reverse()                  Bo
for i in alist:                  Amy
    print(i)                     Amy
for i in reversed(alist):        Bo
    print(i)                     Cy
```

Reverse a word:
```
word = "Python"
iterword = iter(word)
newword =""
for i in reversed(word):
    newword +=i
print (word, newword)
```

range (stop) or **(start, stop [,step])**
```
alist=["Amy","Bo","Cy"]          0 Amy
for i in range (0,len(alist)):   1 Bo
    print(i, alist[i]) #note slice  2 Cy
```

iter and **next(iterator [,default])** Create iterator then fetch next item from iterator. Default returned if iterator exha-usted, otherwise StopIteration raised.
```
alist=["Amy","Bo","Cy"]
IterNum = iter(alist)            Amy
print(next(IterNum, "listend"))  Bo
print(next(IterNum, "listend"))  Cy
print(next(IterNum, "listend"))  listend
print(next(IterNum, "listend"))
```

map(function,iterable) can take multiple iterables but function must take as many
```
alist=[5,9,13,24]
x = lambda z: (z**2 if z**2 < 150 else 0)
itermap = map(x,alist)
for i in alist:
    print(next (itermap))
```
filter(function, iterable) iterator for element of iterable for which function is True.

List Comprehensions

make a new list with item exclusions and modifications from an existing list: brackets around expression followed by 0 to many **for** or **if** clauses; can be nested
Newlst = **[**[modified]item for item in OldLst if some-conditional-item-attribute of (item)**]** or if modifying x only, ex: up1lst =[x+1 for x in ylist]

*args and "kwargs:
are used to pass an unknown number of arguments to a function.
*args is like a list, *kwargs is a keyword = value pair, but keyword cannot be an expression
```
def testargs(a1, *argv):
    print('arg#1: ',a1)
    for ax in range(0,len(argv)):
        print ("arg#"+str(ax+2)+" is "+argv[ax])
testargs('B', 'C', 'T', 'A')
```
⇨ arg#1: B
arg#2 is C
arg#3 is T

```
def testkwargs(arg1, **kwargs):
    print ("formal arg:", arg1)
    for key in kwargs:
        print ((key, kwargs[key]))
testkwargs(arg1=1, arg2="two", dog='cat')
```
⇨ formal arg: 1
('dog', 'cat')
('arg2', 'two')

Useful Modules

Python Standard Library Module Index with links:
https://docs.python.org/3.5/library/
math like Excel math functions ceil (x) .fsum(iterable), sqrt(x), log (x[,base]), pi, e, **random** seed ([x]), choice(seq), randint(a, b), randrange(start, stop [, step]), .random() - floating point [0.0 to 1.0] **sys** stdin standard input, stdout std output, exit (error msg) **datetime** date.today (), datetime.now(), **time** asctime(t), clock(), sleep(secs) **calendar**—a world of date options
```
>>> import calendar
>>> c = calendar.TextCalendar
(calendar.SUNDAY)
>>> c.prmonth(2016, 9)
   September 2016
Su Mo Tu We Th Fr Sa
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30
```

tkinter Python's defacto GUI; also see **ttk; tix**; Older version was: Tkinter (capital T) **os** deep operating system access open(name[,mode[, buffering]]) modes: 'r' reading, 'w' writ-ing, 'a' appending, binary append 'b' like 'rb' **array** work with mathmatical arrays; **tarfile/zip-file** - file compression; **wave** - interface to wav format; **RPi.GPIO** - control Raspberry Pi pins via Python; **csv** import comma sep vals

re-Regular Expresions module
re is a language in itself roughly the size of Python. It supports pattern matching on (1) module-level—for 1 time use and (2) compiled expressions.To compile an expression set a variable like **mypat = re.compile (pattern)** then use it to **search** or **match**. Match searches from first character only. Also you can **findall()** and **finditer().**
import re #if not found 'None'; attribs error
teststring = "Taking Math 101 is fun!"
mypat = re.compile(r'\d+', flags=0)
myso = mypat.search(teststring)
print (myso)
```
print ('group()', myso.group())    ⇨ 101
print ('start()', myso.start())     ⇨ 12
print ('end()', myso.end())         ⇨ 15
print ('span()', myso.span())       ⇨ (12,15)
```
…or don't compile it…
print(re.search(r'\d', teststring).start())
Special characters **. ^ $ * + ? { } [ ] \ ( )**
Use Python r (raw) to process \ commands
r'(pattern)' matches literally: . any except newline \d decimal digit \D non-decimal \w any alphanumeric \W non-alphanum \s any white space chr \S non-whtspace * 0 or more + 1 or more ? 0 or 1 X{n} exactly n ,'X' chars X{m,n} between m & n X's **$** end of str **|** OR: a|b matches a OR b (…) whatever re is in the parens (?abcdef) one or more letters in parens (?=…) a look ahead assertion, "only if" (?!=…) negated look-ahead assertion, "not if" \A match only at start of string \Z match only end of string \b empty string at the start/end of a word [ ] contains a set of chars to match: '-' a range – [a-c] matches a,b,or c special chars lose meaning inside [ ], ^ as 1st char starts complimentary match
Flags: DOTALL any char, A escapes match ASCII, IGNORECASE, MULTILINE affecting ^$, VERBOSE) *About Backslashes:* use the '\' character to indicate special forms or allow a character's use without invoking its special meaning—be aware this conflicts with Python's usage of the same character in string literals.
To test for true result, use bool operator: if bool(re.search(r'\d', teststr))==True: