****
A Tour of Python
author: Phil Pfeiffer
last updated June 2014
****

+++++++++++++++++++++++++
0. Forward
+++++++++++++++++++++++++

This document is intended as a hands-on introduction to Python's chief features for someone with a passing familiarity with a contemporary object-oriented programming language, such as C++, Java, or C#.

Python's primary assets include

-. its widespread adoption by the software community, as attested to by
    -. its ranking at various 'mindshare' websites like www.tiobe.com, and
    -. its use in enterprises like Google and the U.S. Dept. of Energy;
-. its open-source license and extensive set of libraries, making it attractive for development on a tight budget;
-. its support for multiple programming paradigms, including procedural, functional, and object-oriented programming, making it attractive for flexible development and teaching; and
-. its interactive, command-line-based interpreter, which makes it attractive for students-- and developers-- who prefer more rapid feedback than the classic compile-review-re-edit development cycle affords.

This document is specific to Python 3. Python 3, the current major release of Python, eliminated various irregularities in the language while introducing incompatibilities with Python 2. One such change, the replacement of print statements by a built-in print function, renders most logic in this document incompatible with Python 2 and earlier versions.

This document is best viewed in an editor like Notepad++ that supports widescreen text editing. Its nine sections cover the following aspects of the Python language:

-. Obtaining Python.
-. Running Python codes.
-. Python values.
-. Python control structures.
-. Python functions.
-. Python's functional language features.
-. Python classes.
-. OO programming in Python: design considerations.
-. Concluding examples.

Section 2, running Python codes, is currently specific to the Windows environment. While the five strategies for running Python that this section describes have equivalents in the Posix and Mac environments, these equivalents are not discussed here.

Sections 3 through 9 intersperse discussions of Python's features with blocks of code that illustrate those features. These blocks, which are surrounded by comment lines that read

```
############ start of examples ############################################
```
and
```
############  end of examples  ############################################
```
respectively, have been left-aligned in accordance with Python's use of indentation for denoting block structure: a languistic feature that can either be regarded as

-. a nuisance, in that needless or irregular indentation results in syntax errors, or
-. a positive, in that this emphasis on regular indentation yields regular, more concise (i.e., begin-end-free) code.

They are designed to be cut and pasted into a Python interpreter.

This document, though lengthy, is not a complete introduction to Python's feature set. Notable omissions include

- constructs and idioms for creating, structuring, and accessing modules;
- most of the Python library catalogue; and
- metaprogramming in Python, including Python metaclasses.

For more on these and other topics, I recommend the following resources:

- Python's online documentation (cf. www.python.org), particularly
  - The library reference (http://docs.python.org/3/library/), which I consult routinely when writing Python code. The following sections are ones that I've found to be particularly useful:
    - Section 2: builtin functions (http://docs.python.org/3/library/functions.html)
    - Section 4: built-in types (http://docs.python.org/3/library/stdtypes.html)
    - Section 6.2: regular expressions (http://docs.python.org/3/library/re.html)
  - The language reference (http://docs.python.org/3/reference/), particularly
    - Section 3, the Python data model, which features comprehensive documentation Python's special (__...__) attributes (http://docs.python.org/3/reference/datamodel.html).
    - Section 5.2, which documents the structure of Python's modules (packages) (http://docs.python.org/3/reference/import.html#packages)
    - Section 5.3, which documents Python's algorithm for searching modules for content, including how Python's cacheing strategy and sys.modules attribute affect module imports (http://docs.python.org/3/reference/import.html#searching)
  - The Python usage document's section on configuring Python under Windows (http://docs.python.org/3/using/windows.html#configuring-python).
- Mark Lutz's _Learning Python_, 5th edition (O'Reilly), a massive reference (1300+ pages) that covers Python 2 as well as Python 3.
- Mark Summerfield's _Python 3_ (Addison-Wesley), a faster-paced reference that covers much of Python in about 600 pp.

-- Phil Pfeiffer, August 2013

1. Obtaining Python.

Begin by downloading and installing the latest stable release of Python. You can get it from http://www.python.org/download/. As of this writing,

- the current stable release was 3.3.4.
- the link to various implementations of this release, including implementations for Posix, Microsoft, and Mac platforms, was http://www.python.org/download/releases/3.3.4/
- the link to the Windows-64bit precompiled release was http://www.python.org/ftp/python/3.3.4/python-3.3.4.amd64.msi

Install Python to the default install directory. For Python 3.3 under Windows, this is C:\Python33

2. Running Python codes.

This section presents five ways in which Python can be invoked from the Windows environment.

1. As an interactive command line interpreter.

   From the Windows program startup menu,

```
    -.  Locate the Python 3.3 folder.
    -.  In the folder, locate Python (command line)
    -.  Double click on this icon.
        You should see a window entitled "C:\Python33\python.exe"
    -.  Enter the command    1+2    , followed by a return
    -.  Enter the command    exit   , then follow directions.

    Note:  Python uses indentation to delimit blocks.  As a rule, enter all
    commands flush left, unless block structure requires otherwise.

2.-3.  From the Python interactive environment, in two ways.

    2.  From the Windows program startup menu.

    -.  Locate the Python 3.3. folder
    -.  In the folder, locate IDLE (Python GUI)
    -.  Double click on this icon.  This should open a window entitled
        "Python 3.3.2 Shell".  Depending on IDLE's current state, this window
        might be opened as a minimized window, requiring activation from the
        Windows Taskbar.
    -.  Enter the command   1+2     , followed by a return

    3.  From the IDLE environment proper.

    -.  Open a new edit window, in either of two ways:
        -.  by selecting File\New Window
        -.  by entering Control-N.
        Either should open window entitled *Untitled*
    -.  Enter print(1+2) into this window
    -.  Run this code, in one of two ways
        -.  by selecting Run\Run Module
        -.  by typing the F5 key
        IDLE will require you to save this code to a file before running it.
        For this demo, save the code as C:\temp\temp.py
    -.  You should see the result in the window entitled "Python 3.3.2 Shell"

    Finish by closing both windows.

4.-5.  From the command line, in two ways.

   4.  Interactively.

       -.  Raise a Windows command prompt window.
           -.  Note:  you can do this by entering   cmd     into the textbox
               in the Windows startup menu.
       -.  Ensure that the Python home directory is in the command prompt
           window's command path.
           -.  To check if this directory is on the command path,
               -.  Enter the Windows command   set
               -.  Examine the output for the line that starts  Path=
               -.  If this output-- the Path environment variable's string--
                   lacks a reference to the Python installation directory--
                   here, assumed to be C:\Python33\ --
                   -.  Execute the command
                           set Path=%Path%;C:\Python33
                       Important:  do *not* leave spaces around the =
                   -.  Enter the command   set    to confirm that Path now
                       includes  C:\Python33
               Note: the procedure described here affects only the current cmd
               window.  To include (say) C:\Python33\ in every cmd window's
               initial path string, use the Windows control panel's "Advanced
               Settings" option to update Path's default value.
           -.  Enter the command
                    python
               You should now see the Python command prompt.
           -.  Enter  1+2     , followed by a return
           -.  Enter  Control-Z    to exit the interactive session.

   5.  As a command-line command.
```

```
        -.  Raise a command window whose Path environment variable references
            Python's installation directory, as described above.
        -.  Open a file named (say) C:\temp\temp.py in a text editor.
        -.  Edit this file so that it contains one line that reads    print(1+2)
            Position this line flush left in the file.
        -.  From the command prompt, enter the command
                 python  c:\temp\temp.py

            Note:  print(1+2) is needed instead of 1+2 because the interactive
            Python intepreter implicitly redirects the output of top-level
            "expression statements" like 1+2 to the user's console.  In non-
            interactive contexts, printout must be requested explicitly.


+++++++++++++++++++++++++++
3.  Python values.
+++++++++++++++++++++++++++

----------------------------------------------
*.  Illustrating Python support for arithmetic
----------------------------------------------

    Python supports integer, floating point, and complex numbers.
    Python-supported numeric operators include  addition (+), subtraction (-),
    division (/), integer division (//), remainder (%), exponentiation (**),
    bitwise negation (~), bitwise and (&), bitwise or (|),
    bitwise exclusive or (^), bit shift left (<<), and bit shift right (>>).

    Try entering the following:

############# start of examples #############################################

7/3
7//3
7%3
7*3
7**3
7>>1
7<<1

7j**2    # a complex value

# import statements create references to entities in the Python environment.
# range iterators yield a sequence of numbers.
# the "for" statement below consumes those numbers.
# all be covered in more detail later.
#
from math import log10, ceil
for i in range(0, 10000, 100):  print(i, ceil(log10(2**i)), 2**i)

#############  end of examples  #############################################

-.  Illustrating Python support for logical operations.

    Python supports logical and, or, and not, as well as the relational
    operators < (less than), <= (less than or equal), == (equal),
    != (not equal), >= (greater than or equal), and > (greater than).

    As with C, its implementation language, Python treats 0 as False and other
    atomic values as True.  Python also treats empty collections as False
    and non-empty collections as True.

    Try entering the following:

############# start of examples #############################################

# *** ***  illustrating Python's C-like notion of truth  *** ***

'[]  is treated as ' + 'true' if []  else 'false'
'[1] is treated as ' + 'true' if [1] else 'false'
```

```
# **** ***  illustrating Python's support for cascading relational operators  *** ***

4 < 5 < 6
4 < 6 < 5
4 < 6 < 8 > 7 > 5

# **** ***  illustrating floating point imprecision and its management *** ****
#
# idea:
# -.  value_1 captures a series of integer values
# -.  value_2 captures an "equivalent" series or values, generated with floating point math
# -.  as a rule, value_1 != value_2, due to floating point imprecision
# -.  value_2, however, should be "reasonably close" to value_2:
#     i.e., within a factor of epsilon, the smallest distinguishable value between floating point values,
#           scaled by the magnitude of value_1 and value_2

import sys
from math import ceil, log10

print('comparing two sequences of supposedly equal values, generated by integer and floating point
arithmetic, respectively')

low, high, delta = 1000000, 1000099, 0.1

seed_value_2 = delta * low - delta

# Note Python's use of indentation-- rather than paired delimiters like {,} or begin,end--
# to recognize the start and end of the "for" loop's block.
#
# Throughout these codes I follow the Ruby convention for block indentation,
# indenting each block by two spaces from the block that contains it.
#
# Other notes:
# -.   \ at the end of a long line allows for that line's continuation onto the next.
# -.  Python's interpreter uses a single, totally empty line to mark the end of a top-level block.
#
for value_1 in range (low, high):
  seed_value_2 += delta
  value_2 = seed_value_2 / delta
  #
  abs_1, abs_2 = abs(value_1), abs(value_2)
  #
  exact_comparison = value_1 == value_2
  approximate_comparison_tolerance =\
    sys.float_info.epsilon * (1 if min(abs_1, abs_2) == 0 else 10 ** ceil(1 + log10(min(abs_1, abs_2))))
  approximate_comparison = abs(value_2 - value_1) <= approximate_comparison_tolerance
  #
  print( 'values 1 and 2 are {0} and {1}'.format(value_1, value_2) )
  print( 'exact comparison yields {0}; approximate comparison (tolerance: {1}) yields {2}'.\
          format('==' if exact_comparison else '!=', approximate_comparison_tolerance, '==' if
approximate_comparison else '!=') )

#############  end of examples  #############################################
```

```
----------------------------------------------------------
*.  Illustrating Python help conventions (dir, docstrings)
----------------------------------------------------------
```

The following codes illustrate two common mechanisms for creating self-describing Python objects.

The first, the dir() method, when invoked on a Python object, returns a list of that object's attributes, including any references to methods and data that it contains.

The second, a docstring, is a string that's associated with a Python object's '__doc__' attribute. By convention, docstrings describe the purpose and use of their associated objects. They may also specify test

```
    cases for those objects' methods, in ways that support those methods'
    automated testing, via Python library test routines.

    The following codes show a use of dir() statements and docstrings to
    discover the characteristics of Python library objects:  something I often
    do during interactive Python coding sessions.

############# start of examples #############################################

# dir(), when invoked without an argument, returns identifiers in the current environment.
# dir(), at interpreter level, shows identifiers at global scope.
#
if not 'math' in dir():  import math    # make sure 'math' has been imported

dir(math)              # show the routines in the math module,
print(math.__doc__)    # along with the math module's docstring

# show the docstrings of various methods and values from the previous example.
# below, eval(name + '.__doc__') recover the named object's docstring.
# The expression 'eval(..expr..)' interprets the string '..expr..' in the
# current environment, returning the result of this interpretation.
#
from math import *
for name in ['trunc', 'log10', 'ceil', 'degrees']:
  print( eval(name + '.__doc__') + '\n-----\n')

#############  end of examples  #############################################

    By convention, identifiers like __dir__ and __doc__ whose names start
    and end with double underscores are reserved for "special" attributes:
    attributes that

    -.  Python language contructs associate with Python statements, or
    -.  Python library modules associate with certain methods or content

    Here, for example,

    -.  the builtin Python dir() function evaluates 'dir(m)'
        as a call to 'm.__dir__()', while
    -.  Python's doctest library module searches the contents of an object's
        __doc__ attribute for test cases.

    These special attributes provide developers with "hooks" for customizing
    the operation of Python objects to a particular application's needs.


-------------------------------------------------
*.  Illustrating Python built-in data structures
-------------------------------------------------

    In addition to integers and floating point values, Python supports six
    types of native data structures:

    -.  strings
    -.  lists
    -.  dicts
    -.  tuples
    -.  sets
    -.  frozensets

############# start of examples #############################################

#  *********************************************
#  **** **** **** String examples **** **** ****
#  *********************************************

#  *** *** representative examples of strings and string operations *** ***

'abc'
str('abc')     # string object constructor
```

```
len('abc')

'string delimited by \' containing \' and "'
"string delimited by \" containing ' and \""

"""
Initial triple quoting allows a string to cross line boundaries.
The newlines become part of the string.
The string is closed with a matching triple quote: here, ""\"
"""

'''
Similar to the last example, but with ''\' as the opening and closing quotes
and a final character other than \\n'''

r'prepending a single r to a string makes it a "raw" string: one where \ does not escape content'

# if the following content isn't familiar, you missed out at a child.
# see http://ingeb.org/songs/jamesjam.html
#
"{2} {2} {3} {3} {4} {1} {0}".format("Dupree", "George", "James", "Morrison", "Weatherby")
"{action} {adjective} {noun} {preposition} {possessive} {parent}".format(**{'action': 'took',
'adjective': 'great', 'noun': 'care', 'parent':'mother', 'possessive': 'his', 'preposition': 'of'})

'abc' + 'def'
'abc' * 3
'a' in 'abc'
'abc' in 'a'


# *** *** using dir() to learn more about Python strings *** ***

dir(str)    # gives methods for str(ing) objects

print(str.split.__doc__)                # describes the split method, which returns a list of string
fragments
"Took Great Care of his Mother Though he was only three".split()         # again, see http://ingeb.org/
songs/jamesjam.html
"Took Great Care of his Mother Though he was only three".split('e')      # and, if the author's name
isn't familiar, check out his Wikipedia entry

#  ****************************************************
#  **** **** **** String slicing examples **** **** ****
#  ****************************************************

#  **** ****   basic slicing - showing use of Python's [::] operator to extract sample segments of
strings  **** ****

alphabet = 'abcdefghijklmnopqrstuvwxyz'

alphabet[:12]
alphabet[12:]
alphabet[::2]
alphabet[-14:]
alphabet[:-14]
alphabet[::-1]
alphabet[::-2]
alphabet[-14::-2]
alphabet[-2:-14:-2]

#  **** ****   slicing examples, systematized   **** ****

#  *** ***  forward slicing: i.e., slicing with positive stride  *** ***

# -- -- specifying first parameter only:  set substring head; return string tail ----
for i in range(0,len(alphabet)+1):  print("alphabet[{0:>2}:]: {1}".format(i, alphabet[i:]))

# -- -- specifying first parameter only:  illustrating use of "out of range" index values ----
for i in range(-2*len(alphabet),2*len(alphabet)):  print("alphabet[{0:>3}:]: {1}".format(i, alphabet[i:]))
```

```
# -- -- specifying second parameter only:  set substring tail; return string head ----
for j in range(0,len(alphabet)+1):  print("alphabet[:{0:>2}]: {1}".format(j, alphabet[:j]))

# -- -- specifying second parameter only:  illustrating use of "out of range" index values ----
for j in range(-2*len(alphabet),2*len(alphabet)):  print("alphabet[:{0:>3}]: {1}".format(j, alphabet[:j]))

# -- -- specifying third parameter only:  set stride; return slice of string  (note: stride can't be
zero) ----
for k in range(1,len(alphabet)+1):  print("alphabet[::{0:>2}]: {1}".format(k, alphabet[::k]))

# -- -- specifying third parameter only:  illustrating use of out of range stride value ----
print("alphabet[::0]: {1}".format(alphabet[::0]))

# -- -- specifying first and second parameters:  set substring tail, head:  return substring ----
for i in range(0,len(alphabet)+1):
  for j in range(i,len(alphabet)+1):
     print("alphabet[{0:>2}:{1:>2}]: {2}".format(i, j, alphabet[i:j]))    # :>2 specifies 2-wide field,
right aligned value;  see Library reference, section 6.1
  print()

# -- -- specifying first and third parameters:  set substring head, stride:  return slice of string tail
----
for i in range(0,len(alphabet)+1):
  for k in range(1,len(alphabet)+1):
     print("alphabet[{0:>2}::{1:>2}]: {2}".format(i, k, alphabet[i::k]))
  print()

# -- -- specifying first and third parameters:  set substring head, stride:  return slice of string tail
# -- -- break off inner loop when stride grows to where slice is trivial (i.e., 1 char or less) -- ---
for i in range(0,len(alphabet)+1):
  for k in range(1,len(alphabet)+1):
    if len(alphabet[i::k]) > 1:
      print("alphabet[{0:>2}::{1:>6}]: {2}".format(i, k, alphabet[i::k]))
    else:
      print("alphabet[{0:>2}::{1:>2}..{3:>2}]: {2}".format(i, k, alphabet[i::k], len(alphabet)+1))
      break
  print()

# -- -- specifying second and third parameters:  set substring tail, stride:  return slice of string head
# -- -- break off inner loop when stride grows to where slice is trivial (i.e., 1 char or less) -- ---
for j in range(0,len(alphabet)+1):
  for k in range(1,len(alphabet)+1):
    if len(alphabet[:j:k]) > 1:
      print("alphabet[:{0:>2}:{1:>6}]: {2}".format(j, k, alphabet[:j:k]))
    else:
      print("alphabet[:{0:>2}:{1:>2}..{3:>2}]: {2}".format(j, k, alphabet[:j:k], len(alphabet)+1))
      break
  print()

# -- -- specifying first, second, and third parameters:  set string tail, head:  return slice of string
head
# -- -- break off inner loop when stride grows to where slice is trivial (i.e., 1 char or less) -- ---
for i in range(0,len(alphabet)+1):
  for j in range(i,len(alphabet)+1):
    for k in range(1,len(alphabet)+1):
      if len(alphabet[i:j:k]) > 1:
        print("alphabet[{0:>2}:{1:>2}:{2:>6}]: {3}".format(i, j, k, alphabet[i:j:k]))
      else:
        print("alphabet[{0:>2}:{1:>2}:{2:>2}..{4:>2}]: {3}".format(i, j, k, alphabet[i:j:k], len(alphabet)
+1))
        break
    print()
  print('---')


# -- -- illustrating default values of first, second, third parameters -- --
alphabet[::]
alphabet[0:len(alphabet)+1:1]
```

```
#  *** ***  reverse slicing:  i.e., slicing with negative stride  *** ***

# -- -- illustrating default values of first and second parameters with a stride of -1 -- --
alphabet[::-1]
alphabet[-1:-(len(alphabet)+1):-1]


# -- -- first parameter only, stride of -1:  set substring head; return string tail ----
for i in range(0,len(alphabet)+1):  print("alphabet[{0:>2}::-1]: {1}".format(i, alphabet[i::-1]))

# -- -- specifying first parameter only, stride of -1:  illustrating use of "out of range" index values
----
for i in range(-2*len(alphabet),2*len(alphabet)):  print("alphabet[{0:>3}::-1]: {1}".format(i, alphabet
[i::-1]))

# -- -- second parameter only, stride of -1:  set substring tail; return string head ----
# >> notice that you can't get the first character if you specify the second parameter and specify a
stride of -1 <<
for j in range(0,len(alphabet)+1):  print("alphabet[:{0:>2}:-1]: {1}".format(j, alphabet[:j:-1]))

# -- -- specifying second parameter only, stride of -1:  illustrating use of "out of range" index values
----
for j in range(-2*len(alphabet),2*len(alphabet)):  print("alphabet[:{0:>3}:-1]: {1}".format(j, alphabet
[:j:-1]))

# -- -- specifying third parameter only:  set stride; return slice of string  (note: stride can't be
zero) ----
for k in range(1,len(alphabet)+1):  print("alphabet[::{0:>3}]: {1}".format(-k, alphabet[::-k]))

# -- -- specifying first and second parameters:  set substring tail, head:  return substring ----
for i in range(0,len(alphabet)+1):
  for j in range(0,i):
     print("alphabet[{0:>2}:{1:>2}:-1]: {2}".format(i, j, alphabet[i:j:-1]))
  print()

# -- -- specifying first and third parameters:  set substring head, stride:  return slice of string tail
----
# -- -- break off inner loop when stride grows to where slice is trivial (i.e., 1 char or less) -- ---
for i in range(0,len(alphabet)+1):
  for k in range(1,len(alphabet)+1):
    if len(alphabet[i::-k]) > 1:
      print("alphabet[{0:>2}::{1:>8}]: {2}".format(i, -k, alphabet[i::-k]))
    else:
      print("alphabet[{0:>2}::{1:>3}..{3:>3}]: {2}".format(i, -k, alphabet[i::-k], -(len(alphabet)+1)))
      break
  print()

# -- -- specifying second and third parameters:  set substring tail, stride:  return slice of string head
# -- -- break off inner loop when stride grows to where slice is trivial (i.e., 1 char or less) -- ---
for j in range(0,len(alphabet)+1):
  for k in range(1,len(alphabet)+1):
    if len(alphabet[:j:-k]) > 1:
      print("alphabet[:{0:>2}:{1:>8}]: {2}".format(j, -k, alphabet[:j:-k]))
    else:
      print("alphabet[:{0:>2}:{1:>3}..{3:>3}]: {2}".format(j, -k, alphabet[:j:-k], -(len(alphabet)+1)))
      break
  print()

# -- -- specifying first, second, and third parameters:  set string tail, head:  return slice of string
head
# -- -- break off inner loop when stride grows to where slice is trivial (i.e., 1 char or less) -- ---

for i in range(0,len(alphabet)+1):
  for j in range(0,i):
    for k in range(1,len(alphabet)+1):
      if len(alphabet[i:j:-k]) > 1:
        print("alphabet[{0:>2}:{1:>2}:{2:>8}]: {3}".format(i, j, -k, alphabet[i:j:-k]))
      else:
        print("alphabet[{0:>2}:{1:>2}:{2:>3}..{4:>3}]: {3}".format(i, j, -k, alphabet[i:j:-k], len
(alphabet)+1))
```

```
        break
    print()
  print('---')


#  *******************************************
#  **** **** **** List examples **** **** ****
#  *******************************************

#  *** *** representative examples of lists and list operations *** ***

[1, 2, 3]
list([1, 2, 3])       # list object constructor


# *** *** *** operations that characterize lists and/or generate new lists *** *** ***

# --- ---  some (hopefully) familiar operations --- ---

len(['a', 'b', 'c'])
['a', 'b', 'c'] * 3
['a', 'b', 'c'] + ['d', 'e', 'f']


# --- ---  "power" list formers:  list comprehensions   *** ***
#
# expressions that build new lists "on the fly"
# general form of a one-level comprehension:
#    [ value-returning expression  for index variable in sequence of values   if condition holds ]


# - - - - comprehensions involving lists of ints - - - -

[i for i in range(10)]
[[i, j] for i in range(10) for j in range(10) if i < j]

# note that comprehensions can nest

[(i, j) for i in range(10) for j in range(10)]                                    # list of pairs  (1-D
array)

[[(i, j) for i in range(10)] for j in range(10)]                                  # list of list of
pairs  (as 2-D array)
sample_matrix = [[(i, j) for i in range(10)] for j in range(10)]                  # previous expression,
shown more clearly
for item in sample_matrix:  print(item)

[[i * j for i in range(10)] for j in range(10)                     # multiplication table, bare
values     (as 2-D array)
sample_matrix = [[i * j for i in range(10)] for j in range(10)]    # previous expression, shown more
clearly
for item in sample_matrix:  print(item)


# - - - - comprehensions involving lists of strings and characters - - - -

[["{0} * {1} = {2:>2}".format(i, j, i*j) for i in range(10)] for j in range(10)]                    #
multiplication table, with commenting (as 2-D array)
sample_matrix = [["{0} * {1} = {2:>2}".format(i, j, i*j) for i in range(10)] for j in range(10)]    #
previous expression, shown more clearly
for item in sample_matrix:  print(item)

[chr(i) for i in range(ord('a'), ord('z')+1)]
[chr(i) for i in range(ord('a'), ord('z')+1)][:12]
[chr(i) for i in range(ord('a'), ord('z')+1)][-2:-14:-2]

[substr for substr in "Took Great Care of his Mother Though he was only three".split('e') if len(substr)
> 0]    # more whimsey in a good cause
[substr for substr in "Took Great Care of his Mother Though he was only three".split('e') if 'a' in
substr]
```

```
# - - - -  .... and we can generate lists of booleans as well ... - - - -

# substring membership testing

[s in ['a', 'aa', 'aaa'] for s in ['aa', 'aaaa', 'b']]

# type testing

[isinstance(i, str) for i in ['a', 1, 'b', 2, 'c', '3']]
[isinstance(i, int) for i in ['a', 1, 'b', 2, 'c', '3']]

[i for i in ['a', 1, 'b', 2, 'c', '3'] if isinstance(i, str)]

# advanced peek at reduction operations -- meaning, ideally, should be obvious ---

any([isinstance(i, str) for i in ['a', 1, 'b', 2, 'c', '3']])
all([isinstance(i, str) for i in ['a', 1, 'b', 2, 'c', '3']])
all([isinstance(i, str) for i in ['a', 'b', 'c']])



# *** *** representative examples of 'in place' operations on lists *** ***

# these examples require the use of an identifier to maintain a reference to a sample list

x = [1, 2, 3, 4]
x.insert(0, 'a')
x
x.insert(len(x), 'b')
x
x.insert(-1, ['y'])
x
x.insert(-len(x)-1, ['z'])
x
x.reverse()
x
while len(x) > 0: print(x.pop())

x


# *** *** using dir() to learn more about Python lists *** ***

# for convenience, we first define a function, show_object_docstrings(),
# to manage a class's attributes, in a way that accounts for attributes that either
# -.  lack __doc__ attributes, or
# -.  have __doc__ attributes that are set to 'None':  Python's equivalent of SQL's NULL
#
# in what follows,
# -.  eval() raises an exception if the specified attribute isn't present.
# -.  if ... is None: raise Exception raises an exception if a __doc__ attribute is absent
# -.  The "try" block's "except" clause catches these exceptions, reporting that
#     the specified attribute lacks documentation.
# -.  The Python interpreter uses the single blank line after the def statement to
#     recognize where the definition ends and further statements begin.
#
def show_object_docstrings(entity):
  for attr in dir(entity):
    attr_full_name = entity.__name__ + '.' + attr
    print(attr_full_name)
    try:
      entity_docstring = eval(attr_full_name + '.__doc__')
      if entity_docstring is None:  raise Exception
      print(entity_docstring)
    except:
      print('*** no documentation available ***')
    print('-----\n')
```

```
show_object_docstrings(list)


#  ****************************************************************
#  **** **** **** Dict examples, with Tuple examples **** **** ****
#  ****************************************************************

#  *** *** representative examples of dicts and dict operations *** ***

{ 'one': 1, 2.0: 'two point oh' }
dict( { 'one': 1, 2.0: 'two point oh' } )       # dict object constructor
dict( [['one', 1], [2.0,'two point oh']] )      # a list of pairs works as well

{ 'one': 1, 'two': 2, 'three': 3 }['one']
{ 1: 'one', 2: 'two', 3: 'three' }[1]
{ 'one': [1], 'two': [2], 'three': [3] }['one']
{ [1]: 'one', [2]: 'two', [3]: 'three' }[[1]]

#  This last example fails because of Python's requirement that keys for dicts be *immutable*
#  (i.e., unchangeable, constant) values.  This requirement is imposed for efficiency's sake.
#
#  Python, for efficiency, uses hashing on a dict's key to find a key's associated value.
#  Immutability assures that a given value's hash stays fixed over a dict's lifetime.
#
#  Lists, as shown above, are mutable (i.e., updateable).
#      [Note:  this is the reason that list.__hash__.__doc__ is undocumented:  Python treats lists as 
unhashable.]
#  Lists, accordingly, cannot be used as keys for dicts.
#
#  What can be used, instead, is a list-like Python datatype known as a *tuple*.
#  Tuples, like lists, are sequences of values, including (nested) sequences.
#  Unlike lists, tuples are immutable, and can only contain immutable values.

#  *** *** representative examples of tuple operations *** ***

(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
tuple([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])    # tuple object constructor
tuple((1, 2, 3, 4, 5, 6, 7, 8, 9, 10))    # a tuple of values works as well

tuple(i for i in range(10))
tuple(chr(i) for i in range(ord('a'), ord('z')+1))
len((1, 2, 3))

# note that a comma is needed in a singleton tuple to distinguish it from a parenthesized expression
# this is true for strings--a potential gotcha!

type(())
type((3))
type((3,))

for i in ('string',):  print(i)

for i in ('string'):   print(i)


#  *** *** representative examples of dicts with tuples as keys and values *** ***

{ 'one': (1,), 'two': (2,), 'three': (3,) }['one']
{ (1,): 'one', (2,): 'two', (3,): 'three' }[(1,)]

{ (1,): 'one', (2,): 'two', (3,): 'three' }[(4,)]              # this should fail
{ (1,): 'one', (2,): 'two', (3,): 'three' }.get((4,), 'no such value')


x = dict([ ((i, j), i*j) for i in range(10) for j in range(10) if i <= j ])    # using a dict to 
represent a (sparse) matrix
x
for j in range(10):             # again, be careful of the indentation
  for i in range(0, j+1):
    print("{0:3d}".format(x[(i,j)]), end="")    # for more on Python string formatting, see http://
```

```
docs.python.org/3/library/string.html
  print()

#  *** *** more dict methods *** ***

x.keys()
x.items()
x.values()

for i in x.keys(): print(x.pop(i))     # interestingly, this will fail

len({1:2, 3:4})

#  *** *** more about dict and tuple methods *** ***

show_object_docstrings(dict)

show_object_docstrings(tuple)


#  **********************************************
#  **** **** **** Sets and frozensets **** **** ****
#  **********************************************

#  *** *** representative examples of sets and set operations *** ***

type( {3, 4} )     # {....} is used for sets as well as dicts
type( {} )         # {}, however, is interpreted as an empty dict
type( set() )      # this is how to specify an empty set

set( [1, 2] )      # the set() constructor takes one argument:  a sequence of values to setify

# sets, like dicts, are limited to immutables.

set( [[1, 2]] )      #  sets of lists are disallowed...
set( {{1, 2}} )      # ... as are sets of sets ...
set( {{1:2, 3:4}} )  # ... and sets of dicts.

#  *** *** representative examples of set operations *** ***

len({1, 1, 1})
len({1, 2, 3})

{1, 2, 3} | {2, 3, 4}    # union
{1, 2, 3} & {2, 3, 4}    # intersection
{1, 2, 3} - {2, 3, 4}    # difference
{1, 2, 3} ^ {2, 3, 4}    # symmetric difference

{1, 2, 3} > {1, 2}       # test for proper superset
{1, 2, 3} > {1, 2, 3}
{1, 2}    > {1, 2, 3}

{1, 2, 3} >= {1, 2}      # test for superset
{1, 2, 3} >= {1, 2, 3}
{1, 2}    >= {1, 2, 3}

{1, 2, 3} <= {1, 2}      # test for subset
{1, 2, 3} <= {1, 2, 3}
{1, 2}    <= {1, 2, 3}

{1, 2, 3} < {1, 2}       # test for proper subset
{1, 2, 3} < {1, 2, 3}
{1, 2}    < {1, 2, 3}

# *** *** representative examples of 'in place' operations on sets *** ***

# these examples require the use of an identifier to maintain a reference to a sample set

x = {1, 2, 3, 4}
x |= {5, 6, 7}
```

```
x
x &= {1, 2, 3, 4, 5}
x
x -= {4, 5}
x
x ^= {0, 2, 3, 4, 5}
x
while len(x) > 0: print(x.pop())

x

#  *** *** more about set methods *** ***

show_object_docstrings(set)

# *** *** frozensets are like sets, only immutable *** ***

set({frozenset({1, 2})})          # sets can contain frozensets
frozenset({frozenset({1, 2})})    # frozensets can contain frozensets

#############  end of examples  ############################################
```

++

4. Python control structures

++

Basic Python control structures can be divided into three groups:

-. "self-contained" simple statements: "pass", "=" (assignment), "del", "import", "assert", and "raise"
-. compound statements: "if", "while", "for", and "try"
-. simple statements that support compound statements: "break" and "continue"
-. simple statements that qualify scoping: "global" and "nonlocal"

The "global" and "nonlocal" statements will be presented later, in the section on function definition.

----

*. Simple statements

Pass, the simplest of Python's simplest statements, takes no operands and does nothing. It's used primarily as a placeholder, in compound statements and functions that should be included for clarity, but that do nothing.

----

*. Assignment statements

A Python assignment statement updates the value associatd with a Python identifier. Python identifiers consist of

-. an initial letter or underscore, followed by
-. an arbitrary number of letters, numbers, and underscores.

Case is significant: AA, aa, Aa, and aA are different identifiers.

Assignment statements function differently, depending on whether the assignment involves
-. immutable or mutable source objects;
-. referencing, copying, or deepcopying;
-. direct or incremental updating of the target object; and
-. individual or multiple target identifiers

```
############# start of examples ############################################

# *** *** assignments involving immutable objects *** ***
```

```
#
# assignments of immutable values create "private" copies of immutables
# note:  built-in immutables include strings, numbers, frozensets and tuples

x = (1, 2, 3)
y = x
x += (4,)
x
y
x[0] = 0    # this fails
x
y

# *** *** assignments involving mutable objects *** ***

# assignments of mutable values create references to existing copies
# note:  built-in immutables include strings, numbers, frozensets and tuples

x = [1, 2, 3]
y = x
x += [4]
x
y
x[0] = 0
x
y

# Python's copy() built-in creates a "shallow" copy of an object
#
# -.  creating a new top-level structure for the copy, then
# -.  populating this copy with references to the original item's
#     top-level values
#
# Python's deepcopy() built-in creates a complete copy of an object

from copy import copy, deepcopy

x = [0, [2, 3], [[4], [5]]]
y = copy(x)
z = deepcopy(x)
x[0] = 1
x
y
z
x[1][0] = 'two'
x
y
z

# *** *** incremental assignment operators *** ***
#
# Python supports C-style, incremental assignment for various operators, including
#
# +, *, /, //, %, <<, >>,
# & (bitwise and/ set intersection), | (bitwise or/ set union),
# - (subtraction/ set difference), ^ (bitwise xor/ set symmetric difference)

x = 7
x += 1
x
x ^= 7
x
x <<= 2
x

# *** *** concurrent, multi-variable assignment *** ***
#
a, b = 3, 4
a
b
```

```
a, b = b, a      # Python idiom for variable swap
a
b
a, b = 1, 4, 5    # fails: arity must be the same on both sides of assignment

#############  end of examples  ############################################


----------------------------------------------------------------------
*.  Other simple statements: "del", "import", "assert", and "raise"
----------------------------------------------------------------------

    "del", the inverse of assignment, removes identifiers from the Python
    environment.

    "import" imports references to objects in supporting modules into the
    current environment, making the referents available for program use.

    "assert", under ordinary interpreter operation, raises an exception if its
    first operand evaluates to False.  This behavior can be disabled by
    invoking the interpreter with the "optimize" (-O) flag.

    "raise" raises the specified exception.

############# start of examples ############################################

# *** *** "del" statements *** ***

dir()           # shows variables at global scope
'a' in dir()    # should be True
del a           # removes 'a' from global scope
'a' in dir()    # should now be False


# *** *** "import" statements *** ***

'sys' in dir()                                # should be false
import sys                                    # imports a reference to the built-in Python module 'sys'
into the current, global scope
sys.path                                      # shows the sys module's 'path' object.
                                              # this list names the directories from which Python imports
modules.   it can be updated to include more directories
path                                          # path, however, is not in the global scope
from sys import path                          # make sys.path available locally, as path
path                                          # path is now available
module_directories                            # this variable shouldn't be defined yet, either
from sys import path as module_directories   # make path available as module_directories
module_directories                            # should now be available


# *** *** "assert" and "raise" statements *** ***

assert 1==1, 'shouldn\'t throw exception'
assert 1==0, 'should throw exception'

raise LookupError("JAMES JAMES MORRISON'S MOTHER SEEMS TO HAVE BEEN MISLAID.")

#############  end of examples  ############################################


-------------------------------------------------------------------------------
*.  Compound statements:  "if", "for", "while", "try";  "break" and "continue"
-------------------------------------------------------------------------------

    Python's "if', "for", "while", and "try" statements are similar to their
    counterparts in other programming languages.  Key differences are as follows:

    -.  Python uses whitespace to delimit a compound statement's constituent
        blocks, as follows:
```

```
        -.  All statements that are subordinate to a clause in a compound
            statement-- e.g., an "if" statement's "if" and "else" blocks--
            must be indented, relative to their "master" clause.

        -.  All statements that are at the same level in a compound statement
            must have the same initial indentation string.  This second
            requirement can be a nuisance when working with an editor that
            automatically converts tabs to spaces, since Python's "tab nanny"
            treats tabs and spaces as different characters.

        -.  Top-level function and class definitions (see below) must be
            followed by a totally blank line:  i.e., one that contains no
            characters, not even whitespace.  Python's interactive interpreter
            also enforces this requirement for top-level compound statements,
            which is why compound statements in this document are followed by
            blank lines.

    -.  Python's "if" statement, following its "if" clause, supports

        -.  an optional, Modula-like, optional sequence of "elif...elif...elif"
            clauses, followed by

        -.  an optional, final "else" clause.

    -.  Python "for" statements can iterate over sequences of arbitrary values.
        "for" statements are not restricted to integers, as in (say) C.  These
        sequences of values are produced by a kind of state-maintaining function
        called an iterator.  Iterators

        -.  use "yield", rather than "return", to return succesive values
        -.  pick up where left off when invoked from a "for" clause, rather
            than restarting "from scratch"
        -.  execute "return" statements or raise StopIteration exceptions to
            gracefully bring an iteration to a close

    -.  Python "for" and "while" statements have optional "else" clauses, which
        execute once, on normal loop termination.  Using "break" to exit a loop
        bypasses its "else" clause, if present.

    -.  Python "try" blocks that include exception-handling (i.e., "except")
        clauses can be followed by an optional "else" clause.  This clause is
        executed if the body of the "try" clause completes successfully.

    -.  Python "try" blocks can be followed by an optional "finally" clause,
        which always executes after the "try" clause executes, along with any
        of the statement's (optional) "except" and "else" clauses.


############# start of examples #############################################

# *******************************************
# *** *** "if" and "while" statements *** ***
# *******************************************

# showing a function that computes an order-of-magnitude approximation to log10(x)
#
def magnitude(x):
  assert isinstance(x, (int, float)), "value ({0}) is not a number".format(x)
  if x == 0:
    print('value is 0')
  elif abs(x) < 1:
    print('value is between 0 and {0}1'.format('-' if x < 0 else ''))
  else:
    upperbound_log10_x, xcopy, xresidue = '', abs(x), x % 1
    while xcopy > 1:
      xcopy /= 10
      upperbound_log10_x += '0'
    else:                             # just to show the optional "else" clause - could be omitted,
      sign = '-' if x < 0 else ''     # and this block hoisted to the level of the "while"
      if xcopy == 1 and xresidue == 0:
```

```
        print('value is exactly {0}1{1}'.format(sign, upperbound_log10_x))
      else:
        print('value is between {0}1{1} and {0}10{1}'.format(sign, upperbound_log10_x[1:]))

magnitude('a')
magnitude(-30.3)
magnitude(-1)
magnitude(-0.2)
magnitude(0)
magnitude(1)
magnitude(4.2)
magnitude(10)
magnitude(11)
magnitude(20)
magnitude(100)
magnitude(101)
magnitude(594)


# ****************************************************************************
# **** "for" statements, with "break" and "continue" statements and iterators ****
# ****************************************************************************

# example 1:  a straightforward "for" that prints 0..9, all on one line
#
# since the loop contains one statement, this can also be written as
#     for i in range(0,10):  print(i, end=" ")
#
for i in range(0,10):
  print(i, end=" ")


# example 2:  example 1, with the for loop's optional "else" clause used to generate an EOL
#
# since each clause contains one statement, this can also be written as
#     for i in range(0,10):  print(i, end=" ")
#     else:  print()
#
for i in range(0,10):
  print(i, end=" ")
else:
  print()


# example 3:  showing that "break" bypasses "else"
#
for i in range(0,10):
  print(i, end=" ")
  if i == 9:  break
else:
  print()


# example 4:  using an iterator that produces lower-case letters, with a "for" that consumes its values
#
# iterator stops by virtue of "falling off its end" and returning None,
# the default return value when none is specified

# 4.1 - with an index variable in range [ord('a')-1, ord('z')]

def lower_case_letters():
  this_letter = ord('a') -1
  while this_letter < ord('z'):
    this_letter += 1
    yield chr(this_letter)

for lc in lower_case_letters():
  print(lc, end=" ")
else:
  print()
```

```
# 4.2 - with an index variable in range [ord('a'), ord('z')+1]

def lower_case_letters():
  this_letter = ord('a')
  while this_letter <= ord('z'):
    yield chr(this_letter)
    this_letter += 1

for lc in lower_case_letters():
  print(lc, end=" ")
else:
  print()

# 4.3 - with an index variable in range [ord('a'), ord('z')]

def lower_case_letters():
  this_letter = ord('a')
  while this_letter < ord('z'):
    yield chr(this_letter)
    this_letter += 1
  else
    yield chr(this_letter)

for lc in lower_case_letters():
  print(lc, end=" ")
else:
  print()

# 4.4 - with an index variable in range [ord('a'), infinity]
# behaves differently on different systems - but produces incremental output on all

def lower_case_letters():
  this_letter = ord('a')
  while True:
    yield chr(this_letter)
    this_letter += 1

for lc in lower_case_letters():
  print(lc, end=" ")
else:
  print()


# example 5:  like example 4, but using "continue" to skip vowels

def lower_case_consonants():
  this_letter = ord('a') -1
  while this_letter < ord('z'):
    this_letter += 1
    if chr(this_letter) in 'aeiou':  continue
    yield chr(this_letter)

for lc in lower_case_consonants():
  print(lc, end=" ")
else:
  print()


# example 6.1:   fibonacci series iterator with optional count of fib values to yield
# count defaults to continue forever  (i.e., k = infinity)

def fib(val_count = float('inf')):
  if val_count < 1:  raise StopIteration
  yield 0
  if val_count < 2:  raise StopIteration
  prev_2, prev_1, val_count = 0, 1, val_count-2
  yield prev_1
  while val_count > 0:
    val_count -= 1
```

```
    yield prev_2 + prev_1
    prev_2, prev_1 = prev_1, prev_2 + prev_1

[i for i in fib(0)]
[i for i in fib(1)]
[i for i in fib(2)]
[i for i in fib(20)]

#  Example 6.2:  Python should hang on both of these examples

[i for i in fib()]
[i for i in fib() if i <= 1000000000]

#  Example 6.3:  the second example with an explicit cutoff

l = []
for i in fib():
  if i > 1000000000:  break
  l += [i]

l


# example 7
# using "for" statements to modify collections in place can fail
# when the strategy implicitly relies on the collection's initial content

x = [1, 2, 3, 4, 5, 6]
for index in range(0, len(x), 2):
  if index % 2 == 0:  x.pop(index)    # try to remove values at even indices, working forwards

x    # fails, because len(x) is changing as values are removed from x

x = [1, 2, 3, 4, 5, 6]
for index in range(-1, -len(x)-1, -1):
  if index % 2 == 0:  x.pop(index)    # try to remove at even indices, working backwards

x    # also fails

x = [1, 2, 3, 4, 5, 6]
for (index, value) in enumerate(x):
  if index % 2 == 0:  x.pop(index)       # try to remove at even indices, using enumerate

x    # also fails

# here, avoiding incremental update in place works better

x = [1, 2, 3, 4, 5, 6]
x = [value for (index, value) in enumerate(x) if index % 2 == 1]
x


# *******************************
# *** *** "try" statements *** ***
# *******************************

# except clauses can have one of three forms:
# -.  except ..E..:              # ..E.. is class Exception or any of its subclasses
# -.  except ..E.. as ..var..:   # same as above, except ..var.. references the specific Exception object
# -.  except:                    # short for  except Exception:
#
# raise, if executed in an except clause with no arguments, reraises the last exception

# example 1: showing effect of finally, which catches the reraise

def try_test_with_finally(exception_type=None.__class__, message="default"):
  try:
    if issubclass(exception_type, Exception): raise exception_type(message)
  except FloatingPointError as e:
    print("floating point error: {0}".format(e))
```

```
  except ArithmeticError as e:                      # superclass for FloatingPointError
    print("arithmetic error: {0}".format(e))
  except Exception as e:                            # superclass for AritmeticError  (and all exception
classes)
    print("some sort of error: {0}".format(e))
    raise
  else:
    print("no exception happened")
  finally:
    print("ending routine")
    return 'result from finally clause'

print(try_test_with_finally(FloatingPointError, 'suitable floating point error message'))
#-----------
print(try_test_with_finally(ArithmeticError, 'suitable arithmetic error message'))
#-----------
print(try_test_with_finally(OSError, 'suitable OS error message'))    # note that "finally" catches the
re-raise
#-----------
print(try_test_with_finally())
#-----------


# example 2: showing how the try's "return" statement bypasses the "else" clause

def try_test_no_finally(exception_type=None.__class__, message="default"):
  try:
    if issubclass(exception_type, Exception): raise exception_type(message)
    print("ending routine")
  except FloatingPointError as e:
    print("floating point error: {0}".format(e))
  except ArithmeticError as e:                      # superclass for FloatingPointError
    print("arithmetic error: {0}".format(e))
  except Exception as e:                            # superclass for AritmeticError  (and all exception
classes)
    print("some sort of error: {0}".format(e))
    raise
  else:
    print("no exception happened")

print(try_test_no_finally(FloatingPointError, 'suitable floating point error message'))
#-----------
print(try_test_no_finally(ArithmeticError, 'suitable arithmetic error message'))
#-----------
print(try_test_no_finally(OSError, 'suitable OS error message'))
#-----------
print(try_test_no_finally())
#-----------

#############  end of examples  ############################################
```

+++
5. Python functions
+++

Python can be thought of as supporting three kinds of functions:
-. library functions
-. user-defined named functions, like show_object_docstrings()
-. user-defined nameless functions, known as lambda expressions

----
*. Python library functions

    Python's standard library of functions can be divided into two groups:
    -. one group of built-in functions, which are loaded by default into
        the Python environment on Python startup
    -. one group of library functions like math.ceil and math.log10 that must

        be expressly imported into a user's session to be accessed.

-. Built-in library functions.

    Examples of Python built-ins from previous examples include print(), len(), range(), and dir(). The following shows further examples of built-ins.

```
############# start of examples #############################################

# *** *** built-ins for managing sequences:  sorted(), reversed(), zip() *** ***

x = [3, 1, 4, 2]
sorted(x)
reversed(sorted(x))
[i for i in reversed(sorted(x))]
sorted(x, reverse=True)

y = ['three', 'one', 'four', 'two']

zip(x, y)
[v for v in zip(x, y)]


# *** *** built-ins for interpreting strings as code:  eval(), exec()  *** ***

# *** eval: evaluates a single expression, returning a single value ***

eval('3 + 4')

x = 3
eval('x + 4')

eval('x = 3 + 4')   # fails: eval requires expressions, and this is an assignment statement

# *** exec: evaluates a statement block, returning None ***

exec('x = 7; print(x)')
x


# *** *** shows all built-ins *** ***
#
# see http://docs.python.org/3/library/ for additional documentation

import builtins
show_object_docstrings(builtins)       # this is how I learned about sorted()'s reverse keyword

#############  end of examples  #############################################
```

-. Other library functions

    The official documentation for the Python 3 standard library (see http://docs.python.org/3/library/) devotes 28 sections to the non-builtin parts of Python's library. These sections feature logic for

    -. text processing, including regular expressions
    -. binary data processing
    -. auxiliary data types, including temporal types, arrays, and queues
    -. numeric types, including decimal values, fractions, and random values
    -. functional programming
    -. file and directory access
    -. data persistence
    -. data compression and archiving
    -. file formats
    -. cryptography
    -. operating system services
    -. concurrent execution, including threading
    -. interprocess communication and networking

    -. Internet data handling, including e-mail
    -. markup processing, including html and xml
    -. Internet protocols
    -. multimedia
    -. internationalization
    -. program frameworks, including Turtle graphics
    -. GUI interfaces
    -. development tools, including doctest
    -. debugging and profiling
    -. Python runtime services, including sys and futures
    -. custom Python interpreters
    -. Python module importation
    -. language manipulation
    -. generic output formatting
    -. Microsoft-, Unix-, and other-platform-specific modules

    The next example shows functions from Python's file system access module.

```
############# start of examples ############################################

import os, stat, time

# walk a tree in a directory, returning properties of files with a given extension
#
def dirwalk(directory, extension):
  def _onerr(err):
    print("can't access {0}: {1}".format(directory, str(err)))
  try:
    for (this_directory, _, these_filenames) in os.walk(directory, onerror=_onerr):
      files_of_interest = [this_filename for this_filename in these_filenames if '.' + extension ==
os.path.splitext(this_filename)[1]]
      if len(files_of_interest):
        print(this_directory)
        indent = '  '
        for this_file in files_of_interest:
          print(indent, this_file)
          try:
            this_file_fd = os.open(this_directory + '\\' + this_file, os.O_RDONLY)
            try:
              filestat = os.stat(this_file_fd)
              print(3*indent, 'file size:      {0} bytes'.format(filestat.st_size))
              print(3*indent, 'create time:    {0}'.format(time.ctime(filestat.st_ctime)))
              print(3*indent, 'last accessed:  {0}'.format(time.ctime(filestat.st_atime)))
              print(3*indent, 'last modified:  {0}'.format(time.ctime(filestat.st_mtime)))
            except Exception as err:
              print(2*indent, "?? can't stat {0}; {1}".format(this_file, str(err)))
            finally:
              os.close(this_file_fd)
          except Exception as err:
            print(2*indent, "?? can't open {0}; {1}".format(this_file, str(err)))
  except Exception as err:
    print("can't access directory {0}: {1}".format(this_directory, str(err)))

dirwalk('c:\\temp', 'txt')

#############  end of examples  ############################################
```

----
*. User-defined named functions

    Functions, as shown earlier, are defined using Python's def statement.
    "def" is an executable statement that updates Python's environment. The
    following statements apply to Python functions:

    -. Their names must be valid Python identifiers.
    -. They may be defined with arbitrarily many arguments.
    -. They must contain at least one statement.
    -. They may reply to their callers in either of two ways:
        -. They may return values. Functions that return values do not

```
            preserve state between calls.
        -.  They may yield values.  Functions that yield values, known as
            *iterators*, preserve the state of their local data between calls,
            resuming execution at the statement following the last yield.
        By default, a function that "falls off the end" without executing
        a return or a (final) yield is deemed to return the value 'None'.

    Examples of iterators were shown earlier, in connection with "for"
    statements.  The following examples highlight "ordinary", value-returning
    functions.

############ start of examples #############################################

# *** *** a function that takes no arguments and does nothing *** ***
#
# the if and dir() statements in this example are intended to highlight
# the effect of "def" on the interpreter's environment.


if 'f' in dir():  del f     # undefine 'f' if present

'f' in dir()    # confirm that f is absent from the global environment
def f():  pass

'f' in dir()    # 'f' should now be in the global environment
print(f())      # functions that lack returns return the value "None"


# *** **** functions that return values *** ***

def f2():       return 2

def ident(x):   return x

def pair(x,y):  return (x, y)

def k_pairs(v1, v2, k):  return [(v1, v2) for i in range(k)]

f2                    # note that functions are also objects....
f2()

ident
ident(10)
ident(f2)
ident(f2())

pair(3.2, 6)
pair(ident, 4)
pair(ident, 4)[0]       # ... and can be accessed and applied from collections
pair(ident, 4)[0](6)

k_pairs(1, 2, 3)
k_pairs(pair, pair(1, 2), 3)
k_pairs(pair, pair(1, 2), 3)[0][0]
k_pairs(pair, pair(1, 2), 1)[0][0](7, 8)


# *** **** functions can accept and apply other functions as arguments *** ***

def f(g, x, y):
  return g(x, y)

f(pair, 2, 3)


# *** *** more complex examples of function design, using Fibonacci series *** ***

# fibonacci series, version 1:
# efficient fib() with manual memoizing -
# a little clumsy, in that the memoizing (i.e., helper) function is at the top level
```

```
def fib_helper(k):
  if k < 1:  return (0, None, None)
  if k < 2:  return (1, 0, None)
  k_1, k_2, _ = fib_helper(k-1)
  return (k_1 + k_2, k_1, k_2)

def fib(k): return fib_helper(k)[0]

[fib(i) for i in range(36)]


# fibonacci series, version 2:
# efficient fib() with manual memoizing as a local helper function

# undefine top-level fib_helper for purpose of illustration
#
fib_helper
globals().pop('fib_helper')
fib_helper

def fib(k):
  def fib_helper(k):
    if k < 1:  return (0, None, None)
    if k < 2:  return (1, 0, None)
    k_1, k_2, _ = fib_helper(k-1)
    return (k_1 + k_2, k_1, k_2)
  return fib_helper(k)[0]

fib_helper

[fib(i) for i in range(36)]


# fibonacci series, version 3:
# compact but inefficient (exponential) fib() with no memoization

def fib(k):  return 0 if k < 1 else 1 if k < 2 else fib(k-1) + fib(k-2)

[fib(i) for i in range(36)]


# fibonacci series, version 4:
# uses Python functional that wraps fib() in logic that memoizes intermediate results

from functools import lru_cache

@lru_cache(maxsize=None)
def fib(k):  return 0 if k < 1 else 1 if k < 2 else fib(k-1) + fib(k-2)

[fib(i) for i in range(36)]


# **** examples of function closures ***
#
# a simple closure; it retains one value, which remains constant

def plus_n(n):
  def f(x): return n+x
  return f

plus_4 = plus_n(4)

plus_4(-4)
plus_4(0)
plus_4(3)


# a function that updates its closure
#
```

```
# note that functions can hold attributes:
# here, a reference to a local attribute that f() uses to compute its result

def f():
  f.x = 0 if not(hasattr(f, 'x')) else f.x+1
  return f.x

f()
f()
f()


# a self-destructing function  (not recommended, as a rule...)  ****

def f():
  global f
  del f
  return "this could be the last time, \n"*2 + "maybe the last time, I don\'t know\n"

print(f())
print(f())

#############  end of examples  ###########################################


------------------------------------------------------------
*.  Function definitions, environment, and scoping rules
------------------------------------------------------------

    Python's def statement dynamically alters Python's execution environment:
    i.e., the set of lists that Python searches to determine the values of
    identifiers in the expressions it executes.

    A def statement only affects Python's environment if it successfully
    completes execution.  A def with a malformed header clause or body-- one
    that won't parse or violates one of Python's rules for well-formedness--
    has no effect:  it simply leaves the environment unchanged.

    To understand exactly how a successful def statement affects the Python
    environment, it helps to understand how Python's environment is structured.

    I think of the Python environment as an object that contains
    -.  a master list of all identifiers that are defined for statements that
        are
        -.  positioned outside of all function and class definitions:
        i.e., at the interpreter's global scope.

    -.  each of the function and class objects in this master list, in its
        turn, contains a master list of identifiers that are defined for
        statements that are
        -.  positioned inside that function or class, but
        -.  outside of all other functions or classes that this object
            might contain;
        i.e., at a scope depth of 1.

    -.  each of the functions and class objects in these scope-level-1 lists,
        in its turn, contains a master list of identifiers that are defined for
        statements that are
        -.  positioned inside that function or class, but
        -.  outside of all other functions or classes that this object
            might contain;
        i.e., at a scope depth of 2.

    This nesting of "code container" objects within "code container" objects--
    e.g.,

    -.  functions within the outermost Python interpreter
    -.  functions within functions within the outermost Python interpreter
    -.  functions with classes within the outermost Python interpreter
    -.  functions within functions within classes within the outermost Python
```

```
        interpreter
    -.  functions within functions within classes within classes within the
        outermost Python interpreter

    can be continued indefinitely.  The resulting nesting of code containers
    can yield a nesting of definitions that's similar to the nesting common
    to common, lexically scoped languages like Pascal, C, C++, Java, and C#.

    One difference between Python and these other languages is that Python
    "while", "for", "try", and "if" statements don't open new scopes, as is
    true in (say) C.   Within any given Python module, only def and class
    statements create new scopes.

    A second difference is due to Python's lack of mandatory declarations.
    In most statically scoped 3GL's, the scope to which a nonlocal identifier
    x belongs is determined by looking "upward and outward" from a use of x,
    until a declaration for x is encountered.

    Since Python, by default, doesn't support variable declarations, this
    "upward and outward" strategy can't be used to disambiguate references.
    What Python uses, instead, is a set of rules that, as I understand them,
    work as follows:

    *.  Python essentially supports three scopes:
        -.  global scope, the scope of the top-level interpreter
        -.  local scope, the scope of current def or class declaration
        -.  nonlocal scope, the scope of any identifiers that lie "above"
            the current def or class declaration but "below" global scope.

    *.  Python allows you to declare identifiers as global or nonlocal.  Python
        has no "local" declaration:  variables that are determined to be local
        are done so, by default, in ways that make an initial assignment to a
        variable local, by default.

    *.  The rules for identifier scoping, as I understand them, are as follows:

        -.  All identifiers defined at Python's global scope are global.
            -.  global declarations are ignored at the global level, while
            -.  nonlocal declarations are treated as erroneous

        -.  Any identifier may be the subject of at most one of three
            declarations at the top level of any function or class:

            -.  a global declaration
            -.  a nonlocal declaration
            -.  a formal parameter declaration

            Any definition that contains two or more of these declarations
            for the same identifier at the top level of a given function or
            class is malformed.

        -.  For an identifier x that is named at most by one declaration,
            the following decision process determines x's scope:
            -.  is x is named in a global declaration?
                 -.  yes:  x is treated as a global (top-scope) value.
                 -.  no:   is x named in a nonlocal declaration?
                     -.    yes:  x is treated as a nonlocal value.
                     -.    no:  is x named as a formal function parameter?
                           -.    yes:  x is treated as a local value that, if bound to a mutable,
                                       references a value defined elsewhere
                           -.    no:   is x read before being updated in the current scope?
                                 -.    yes:  x is read from the closest enclosing scope
                                 -.    no:   x is treated as a local value

    The following examples illustrate these rules.

############# start of examples ############################################

# *******************************************************
# *** *** examples of globally scoped identifiers *** ***
```

```
# *******************************************************

x = 'global value'

def f():   # example global_1:  g() returns x's global value rather than its nonlocal value (0)
  x = 'nonlocal value for global_1'
  def g():
    global x
    return x
  print('global_1 definition')
  return g()

f()  # note that the nonlocal value for x is ignored
#-------------------------------

def f():    # example global_2:  Python flags this definition as erroneous, due to the naming conflict in
g()
  x = 'nonlocal value for global_2'
  def g(x):
    global x
    return x
  print('global_2 definition')
  return g(x)

f()  # since the definition of f() above is rejected, the first (example 1) definition stays active
#-------------------------------

def f():     # example global_3:  here, h() returns x's global value rather than its nonlocal value
  x = 'nonlocal value for global_3'
  def g(x):
    def h():
      global x
      return x
    return h()
  print('example global_3 definition')
  return g(x)

f()  # note that the nonlocal value for x is ignored
#-------------------------------

def f():     # example global_4:  here, x references a global, since it's read before being written
             # and not defined in any enclosing scope
  def g():
    return x
  print('example global_4 definition')
  return g()

f()
#=====================================


# *********************************************************
# *** *** examples of nonlocally scoped identifiers *** ***
# *********************************************************

def f():    # example nonlocal_1:  this is valid, since x is "homed" in f's outermost scope
  x = 'outer nonlocal value for nonlocal_1'
  def g():
    nonlocal x
    x = 'inner nonlocal value for nonlocal_1'
  print('example nonlocal_1 definition')
  g()
  return x

f()
#-------------------------------

def f():    # example nonlocal_2:  similar to nonlocal_1, since x is defined in f()
  x = 'nonlocal value for nonlocal_2'
  def g():
```

```
    return x
  print('example nonlocal_2 definition')
  return g()

f()
#-------------------------------

def f():   # example nonlocal_3:  the following is not valid, since x is not "homed" in any nonglobal
scope
  def g():
    nonlocal x
    x = 'nonlocal_value for nonlocal_3'
    return x
  print('example nonlocal_3 definition')
  return g()

f()   # note that example nonlocal_2 definition is still active
#-------------------------------

def f(x):   # example nonlocal_4:  the following is valid, since x is "homed" in f's declaration
  def g():
    return x
  print('example nonlocal_4 definition')
  return g()

f('parameter value')
#===================================


# *******************************************************
# *** *** examples of locally scoped identifiers *** ***
# *******************************************************

def f():   # example local_1:  the following is valid, since x is defined before use
  def g():
    x = 'local value'
    return x
  print('example local_1 definition')
  return g()

f()
#-------------------------------


def f():   # example local_2:  the following is treated as valid, since x is referenced (via the +=)
before use
  def g():
    x += 'local value'
    return x
  print('example local_2 definition')
  return g()

f()
#-------------------------------

def f():   # example local_3:  the following is valid
  def g():
    x = 'local'
    x += ' value'
    return x
  print('example local_3 definition')
  return g()

f()

############  end of examples  ############################################
```

----
*. Varargs and keyword arguments

    Python doesn't support overloading in the C++/ Java/ C# sense of the term.
    If you create a second definition a function f in a given scope, the
    second definition overwrites the first. Functions, however, can be made
    to accept varying numbers of values-- a common goal of overloading--
    using keywords and varargs.

    The following examples illustrate this point.

```
############# start of examples #############################################

# ********************************************************
# **** ****  examples of varargs functions  **** ****
# ********************************************************

def f(*args):              # just varargs.  max: one varargs param per function
  for arg in args:         # args is returned to the function as a list
     print("arg is {0}".format(arg))

f(1, 2, 3, 'a', 'b', 'c', [4, 5, 6])

arg_list = [1, 2, 3, 'a', 'b', 'c', [4, 5, 6]]
f(*arg_list)   # can also pass a list as a keyword argument, using an initial *
f(arg_list)    # notice what happens when * is omitted -- one argument


def f(*args):                            # again, just varargs
  for (argno, arg) in enumerate(args):   # enumerate returns (position number, value) pairs
     print("vararg #{0} is {1}".format(argno, arg))

f(1, 2, 3, 'a', 'b', 'c', [4, 5, 6])
f(*arg_list)   # can also pass a list as a keyword argument, using an initial *
f(arg_list)    # notice what happens when * is omitted -- one argument


def f(a, b, c, *args):                # varargs mixed with ordinary args.  varargs must follow ordinary
args
  print("a, b, c are {0}, {1}, {2}".format(a, b, c))
  for (argno, arg) in enumerate(args):
     print("vararg #{0} is {1}".format(argno, arg))

f(1, 2, 3, 'a', 'b', 'c', [4, 5, 6])   # succeeds
f(1, 2, 3)                             # succeeds, but with no varargs
f(1, 2)                                # fails - insufficient number of args


def f(a, b, c=17, *args):              # varargs mixed with ordinary args and a parameter, c, with a
default value
  print("a, b, c are {0}, {1}, {2}".format(a, b, c))
  for (argno, arg) in enumerate(args):
     print("vararg #{0} is {1}".format(argno, arg))

f(1, 2, 3, 'a', 'b', 'c', [4, 5, 6])   # succeeds
f(1, 2, 3)                             # succeeds, but with no varargs
f(1, 2)                                # succeeds - again with no varargs


# ********************************************************
# **** ****  example of keyword functions  **** ****
# ********************************************************

def f(**kwds):                           # just keywords.  max: one keywords param per function
  for (key, value) in kwds.items():      # kwds is returned to the function as a dict
    print("keyword {0} is {1}".format(key, value))

f(first=1, second=[2,2], third=(3,), fourth={4:'four', 'four':4})    # keyword keys must be valid
identifiers
```

```
kwd_dict = {'first':1, 'second':[2,2], 'third':(3,), 'fourth':{4:'four', 'four':4}}
f(**kwd_dict)   # can also pass a dict as a keyword argument, using an initial **


# ***********************************************************
# **** ****  combining varargs and keywords  **** ****
# ***********************************************************

# note that varargs must precede keywords:  this is required

def f(a, *args, **kwds):                   # combining all three
  print("a is {0}".format(a))
  for (argno, arg) in enumerate(args):
     print("vararg #{0} is {1}".format(argno, arg))
  for (key, value) in kwds.items():        # kwds is returned to the function as a dict
    print("keyword {0} is {1}".format(key, value))

arg_list = [1, 2, 3, 'a', 'b', 'c', [4, 5, 6]]
kwd_dict = {'first':1, 'second':[2,2], 'third':(3,), 'fourth':{4:'four', 'four':4}}

f(1, *arg_list, **kwd_dict)
f(1, **kwd_dict, *arg_list)    # will fail
f(1, **kwd_dict)               # will succeed - no varargs
f(1, *arg_list)                # will succeed - no keywords

#############  end of examples  #############################################
```

```
--------------------------
*.  lambda expressions
--------------------------
```

A lambda expression-- so named because its originator, mathematician Alonzo Church-- used the Greek letter "l" as a shorthand for "let"-- is a nameless function that consists of

-. the keyword "lambda", followed by
-. a list of the function's arguments, followed by
-. the function's body: a single expression

Lambdas were supported in one of the first widely used higher level programming languages, the functional programming language Lisp. Lambdas have been a staple of functional programming languages for decades. More recently, lambdas have been incorporated into a "mainstream" programming language, LINQ.

```
############# start of examples #############################################

# *** a bare lambda that isn't applied to any values - interesting, but useless ***

lambda x, y: x+y


# *** that lambda applied to some values ***

(lambda x, y: x+y)(3, 4)
(lambda x, y: x+y)('a', 'b')
(lambda x, y: (x, x, y, y))(3, 4)
(lambda x, y: (x, x, y, y))('a', 'b')

(lambda x, y: x=y)('a', 'b')    # fails - assignment isn't an expression
(lambda x, y: )('a', 'b')       # fails - must have an expression to the right of the colon
(lambda x, y: None)('a', 'b')   # succeeds - does nothing


# *** that lambda, bound to a name, then applied by name to some values ***

f = lambda x, y: x+y
f(3, 4)
f('a', 'b')
```

```
# *** a lambda with no arguments ***

f = lambda: 3
f()


# *** lambdas with default values, varargs lambdas, and keyword lambdas ***

f = lambda x=3: x
f(4)
f()

f = lambda *args: [i for i in args if i > 0]
f(4, -3, 2, -1, 0)

f = lambda **kwds: {i:j for (i,j) in kwds.items() if i.startswith('a')}
f(apple=1, pear=2, aardvark=3, parrot=4)


# *** a recursive lambda ***

sum_between = lambda start, finish:  0 if start > finish else start + sum_between(start+1, finish)
sum_between(0, 4)
sum_between(4, 0)
sum_between(-4, 4)
sum_between(20, 30)


# *** using the sequence (,) operator to do multiple actions in a lambda

# first example - lambda with no side effects

f = lambda x: (print(x+1), print(x+2), print('-----'), x)[3]

f(5)

# second example - exec() with globals() used to install identifiers in Python's global environment

def assign(x, y):
  exec(x + '=' + y, globals())

f = lambda w, x, y, z:  (assign(w, x), assign(y, z), (w, y))[2]

f('a', '4', 'b', '5')
a
b


# *** a lambda with a comprehension

take = lambda l, filter:  [ value for (index, value) in enumerate(l) if filter[index] ]

# "take every nth" is simpler to code as a slicing operation.  it's shown here for purposes of
illustration.

take_every_nth  =  lambda l, n, offset: take( l, [ (i - offset)%n == 0 for i in range(len(l)) ] )
take_every_nth( [i for i in range(10)], 3, 0)
take_every_nth( [i for i in range(10)], 3, 1)
take_every_nth( [i for i in range(10)], 3, 2)

# "drop every nth", on the other hand, can't be coded simply using slices

drop_every_nth  =  lambda l, n, offset: take( l, [ (i - offset)%n != 0 for i in range(len(l)) ] )
drop_every_nth( [i for i in range(10)], 3, 0)
drop_every_nth( [i for i in range(10)], 3, 1)
drop_every_nth( [i for i in range(10)], 3, 2)

#############  end of examples  #############################################
```

```
+++++++++++++++++++++++++++++++++++++++++++++++++++++
6.  Python's functional programming features
+++++++++++++++++++++++++++++++++++++++++++++++++++++

-------------------------------------------------------------
*.  About functional (i.e., sequence-oriented) programming
-------------------------------------------------------------

Functional programming, loosely speaking, is computation with operators that
-.  take sequences as inputs, returning
-.  new sequences or values as outputs

Functional programming typically also includes functionals:  functions that
-.  takes functions as inputs, returning
-.  (related) functions as outputs

Functional programming's chief virtue is the concision afforded by the use of
sequence-based operators and functions in place of explicit loops.  The
resulting, shorter codes make logic easier to write, read, and maintain.

Earlier examples have highlighted four functional programming constructs:
-.  slicing - i.e., the use of [x:y:z] style syntax to subset sequences
-.  list comprehensions - i.e., the use of [... for ... in ....]
    to construct lists
-.  filters - i.e., the use of "if ..." clauses in comprehensions to limit
    a comprehension's content
-.  lambdas - i.e., nameless functions

A fourth useful construct, reduction, is highlighted below.  Reduction is
analogous to database aggregation, but with a parameterized aggregation operator.
Python's reduction function-- actually, a functional-- takes three arguments:

-.  a function that
    -.  accepts two arguments:
        -.  a value that represents an intermediate result
        -.  a value that represents the next value in a sequence to reduce
    -.  returns the result of continuing the reduction, using these two inputs
-.  a sequence to reduce
-.  an initial value, for priming the intermediate result.
    This initial value is typically the identify element for the
    operation that's driving the reduction.

Reductions can also be conceptualized in terms of accumulator loops, which they
typically replace.  Intuitively,

-.  the function's first parameter corresponds to an accumulator variable
-.  the function's second parameter corresponds to the loop's index variable
-.  the initial value corresponds to the accumulator variable's initial
    value, set outside the loop.


############# start of examples #############################################

# *** *** some builtins for functional-style logical operations and their equivalent reductions *** ***

any( [ ] )
any( [False, False] )
any( [True, False]  )
any( [True, True]   )
all( [ ] )
all( [False, False] )
all( [True, False]  )
all( [True, True]   )

from functools import reduce

reduce( lambda so_far, next: so_far or  next, [ ], False )                 # what any() looks like as a
reduction
```

```
reduce( lambda so_far, next: so_far or  next, [ False, False ], False )     # False is the identity
element for logical 'or'
reduce( lambda so_far, next: so_far or  next, [ True, False ],  False )
reduce( lambda so_far, next: so_far or  next, [ True, True ],   False )
reduce( lambda so_far, next: so_far and next, [ ], True  )                  # what all() looks like as a
reduction
reduce( lambda so_far, next: so_far and next, [ False, False ], True  )     # True is the identity
element for logical 'and'
reduce( lambda so_far, next: so_far and next, [ True, False ],  True  )
reduce( lambda so_far, next: so_far and next, [ True, True ],   True  )

# *** *** some builtins for functional-style numeric operations and their equivalent reductions *** ***

sum( [ ] )
sum( [ ] , 5 )
sum( [1, 2, 3, 4] )
sum( [1, 2, 3, 4], 5 )

reduce( lambda so_far, next: so_far + next, [ ], 0 )               # what sum() looks like as a reduction
reduce( lambda so_far, next: so_far + next, [ ], 5 )
reduce( lambda so_far, next: so_far + next, [1, 2, 3, 4], 0 )
reduce( lambda so_far, next: so_far + next, [1, 2, 3, 4], 5 )

my_sum = lambda l, identity_element=0: reduce( lambda so_far, next: so_far + next, l, identity_element
)    # defining sum() using a lambda
my_sum([])
my_sum([], 5)
my_sum([1, 2, 3, 4])
my_sum([1, 2, 3, 4], 5)

min( [-100, 40, 90, 700] )
max( [-100, 40, 90, 700] )

reduce( lambda so_far, next: so_far if so_far <= next else next, [-100, 40, 90, 700], float('inf') )
# what min() looks like as a reduction
reduce( lambda so_far, next: so_far if so_far >= next else next, [-100, 40, 90, 700], float('-inf') )
# what max() looks like as a reduction

# note that positive and negative infinity are the identity elements for min() and max(), respectively


# *** *** a few more "interesting" reductions *** ***

# *** showing the use of + to concatenate ***

my_sum( [chr(i) for i in range(ord('a'), ord('z')+1)], '' )
my_sum( [chr(i) for i in range(ord('z'), ord('a')-1, -1)], '' )

# *** the 'identity' reduction for lists
#
# while this reduction essentially does nothing, it's a good starting point for developing more complex
"reductions" over lists

list_identity = lambda l: reduce (lambda so_far, next: so_far + [next], l, [])

# *** I've given the next reduction, which is kind of an anti-reduction, as an assignment.
#
# note the relationship between this and the previous list_identity reduction

expand = lambda l, gapcounts, gap0count, null: \
                reduce( lambda so_far, next: so_far + [next[0]] + next[1]*[null], zip(l, gapcounts),
gap0count*[null])
expand(['a', 'b', 'c', 'd'], [1, 2, 0, 1], 3, None)

# Functions that drop elements from lists can also be implemented as reductions.
# I've found it cleaner, however, to frame these using comprehensions with filters.
# See the take and drop examples, shown earlier.

#############  end of examples  #############################################
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++
7.  Python classes
++++++++++++++++++++++++++++++++++++++++++++++++++++

------------------
*.  Introduction
------------------

    A Python class can be regarded as a **unique** **container** whose
    **content** **can be used** to **generate** a **family** of **objects**:

    -.  **unique**.  Each Python class has an identity that differentiates it
        from all other objects in a Python environment.

    -.  **container**.  A Python class is structured as a list of references
        to values.  These references are commonly referred to as a class's
        **attributes**.

    -.  **content**.  A class's attributes are said to belong to that class,
        in the sense that a Python associates a class's name with these
        attributes.

    -.  **can be used**.  Some Python statements and operators invoke certain
        "special" attributes with well-known names to accomplish class-related
        actions.  For example,

        -.  Python's dir() built-in invokes a class's __dir__ attribute to
            list that class's attributes.
        -.  Python's id() built-in invokes class's __hash__ attribute
            to retrieve that class's unique identifier.

    -.  **generate**.  Two of a class's special attributes, __new__ and
        __init__, are invoked by Python's constructor construct to create
        and initialize a new object.

    -.  **family**.  Python treats a class as a type.  All objects that are
        are generated from that class initially have that class as their type.
        (Note: the word *initially* is used here because a Python code can,
         subject to certain restrictions, dynamically change an object's type.)

    -.  **object**.  An object, according to Page-Jones, is an entity that
        has the following five characteristics:

        -.  an *identity*:
               a value that distinguishes this object from other objects in
               the Python environment
        -.  a *state*:
               a collection of (subordinate) objects that are associated
               with (i.e., "internal to") that object and that can vary
               over the course of a computation
        -.  an *interface*:
               a set of identifiers that reference the object's contents.
               this includes *methods*:  actions that are associated with that
               object that can access and possibly update an object's state
        -.  a set of *behaviors*:
               actions that affect itself and its environment,
               as implemented by its methods.
        -.  a *lifetime*:
               a duration in an overall computation during which the
               object exists.

        In addition to these five characteristics, Python objects, as noted
        above, are typed.

    In addition to these qualities, Python classes are also subclassed.  Every
    Python class is a subclass of at least one other class:  possibly, of two
    or many.  Ultimately, all classes are descended from class object, the
    class at the top of the Python class hierarchy.  As in other object-
    oriented (OO) languages, saying that a Python class B is a subclass of a
```

```
    Python class A means that class B can potentially *inherit* A's content:
    i.e., access and manipulate A's content directly, as if that content were
    part of B's explicit definition.

    The following examples illustrate these basic principles of Python classes.


############# start of examples #############################################

# classes are created by an executable compound statement, class.  It consists of two parts:
# -.  a header that names the class to create, and (optionally) its superclasses
# -.  a body that specifies the class's initial content

class Trivial:  pass

# *** *** as a class, Trivial has a unique (if uninteresting) id *** ***

id(Trivial)

# *** *** if a class's superclasses not specified, it is assigned one superclass: class object *** ***

Trivial.__class__.__base__        # shows a class's immediate superclass

# *** *** Trivial, as a subclass of object, inherits object's attributes *** ***

dir(object)
dir(Trivial)
set(dir(object)) - set(dir(Trivial))    # showing that every attribute in object is in Trivial
set(dir(Trivial)) - set(dir(object))    # showing that Trivial adds a few attributes to object

# *** *** Trivial can be used to generate new objects *** ***

trivial_instance_1 = Trivial()
trivial_instance_2 = Trivial()

# *** *** these objects are both of type Trivial *** ***

type(trivial_instance_1)
type(trivial_instance_2)

# *** *** they are, however, distinct from Trivial as well as each other *** ***

id(Trivial)
id(trivial_instance_1)
id(trivial_instance_2)

id(Trivial) == id(trivial_instance_1)
id(Trivial) == id(trivial_instance_2)
id(trivial_instance_1) == id(trivial_instance_2)

############# end of examples ###########################################

--------------------------------------------------------
*.  Instance methods, staticmethods, and classmethods
--------------------------------------------------------

    When a class constructs an instance of itself, it associates that instance
    with a directory of references that Python statements and operators use to
    accomplish instance-related actions.  These references can be divided into
    two kinds of references:

    -.  references to shared entities.  All instances of a class share the code
        objects that correspond to the methods that a class defines.  All
        instances of a class also share data items that are local to the class
        definition.

    -.  references to entities that are specific to the instance.  Every
        instance of a class is initialized with a reference to a set of
        references that are exclusive to that class instance.  By convention,
        that reference to an instance's private data is called "self".
```

```
    The methods that a class contains can be divided into three basic categories,
    according to the objects that they access.

    -.  instance methods.   An **instance method** is one that can access self:
        i.e., the data associated with a **specific** instance of a class.

    -.  staticmethods.  A **staticmethod** is limited to accessing references
        to a class's shared entities:  i.e., entities associated with the class
        proper.  Staticmethods are commonly used to manage a class hierarchy's
        shared instance data:  i.e., data that is essentially hosted by the
        hierarchy as a whole, rather than any one class in the hierarchy.

    -.  classmethods.  A **classmethod** is essentially a staticmethod that
        also has access to the contents of the class that instantiated its
        associated object.  Classmethods are commonly used to manage content
        that is specific to a class:  i.e., data that is essentially hosted by
        a specific class, rather than an inheritance hierarchy as a whole.

    The distinction between staticmethods and classmethods is subtle, and,
    from what I can tell, probably immaterial for most applications.  I show
    it here, however, for completeness.

    Instance methods, staticmethods, and classmethods are defined in
    different ways:

    -.  Instance methods include a special first parameter, typically called
        self, that references the instance's private data.  All instance data
        should be stored in and accessed from self.

    -.  Staticmethods are defined without this special first parameter.
        A method intended for use as a staticmethod, however, must be wrapped
        in special logic that preprocesses its inputs.  This is done by setting
        the desired staticmethod name to the output of a built-in Python
        functional, staticmethod(), that does the wrapping.

    -.  Classmethods are defined with a different special first parameter:
        the object of the class that instantiated the current instance.
        A method intended for use as a classmethod must also be wrapped
        in special logic that preprocesses its inputs.  This is done by setting
        the desired classmethod name to the output of a built-in Python
        functional, classmethod(), that does the wrapping.

    The examples that follow illustrate Python protocols for invoking instance
    methods, staticmethods, and classmethods as well as the differences between
    these types of methods.

############ start of examples ###########################################

# *******************************************************************************************
# *** *** construct a class with instance methods, staticmethods, and classmethods,  *** ***
# *** *** together with a representative subclass                                    *******
# *******************************************************************************************

class MyClass:
  # ---- these next six lines define two staticmethods for this class
  #
  def set_static_value(newv):                        # these 2 lines define a setter for MyClass's
static_value variable
    MyClass.static_value = newv
  set_static_value = staticmethod(set_static_value)   # this line redefines set_static_value as a
staticmethod
  def get_static_value():                            # these 2 lines define a getter for MyClass's
static_value variable
    return MyClass.static_value
  get_static_value = staticmethod(get_static_value)   # this line redefines get_static_value as a
staticmethod
  #
  # ---- these next six lines define two classmethods for this class
  #
```

```
  def set_classattr_foo(thisclass, value):            # these 3 lines define a classmethod for MyClass
that sets an arbitrary attribute
    thisclass.foo = value
  set_classattr_foo = classmethod(set_classattr_foo)  # required to make set_classattr a classmethod
  def get_classattr_foo(thisclass):                   # these 3 lines define a classmethod for MyClass
    return thisclass.foo
  get_classattr_foo = classmethod(get_classattr_foo)  # required to make get_classattrs a classmethod
  #
  # ---- these final four lines define two instance methods for MyClass
  #
  def set_instance_value(self, v):                    # these 2 lines define an instance method for
MyClass
    self.instance_value = v
  def get_instance_value(self):                       # these 2 lines define an instance method for
MyClass
    return self.instance_value

# *** *** create a trivial subclass of this class to illustrate the methods' operation

class MySubclass(MyClass): pass

# **********************************************************************************************
# *** *** show Python protocls for invoking instance methods, staticmethods, and classmethods  *** ***
# **********************************************************************************************

# *** *** create instances of these classes to illustrate the methods' operation

myclass_instance_1 = MyClass()
myclass_instance_2 = MyClass()
mysubclass_instance = MySubclass()

# *** *** show that the instance have types that correspond to their classes

type(myclass_instance_1)
type(myclass_instance_2)
type(mysubclass_instance)

# *** *** illustrate instance method invocation and operation

myclass_instance_1.set_instance_value(1)            # - - - instance methods are commonly invoked by
class instance
myclass_instance_2.set_instance_value(2)
mysubclass_instance.set_instance_value(3)

myclass_instance_1.get_instance_value()
myclass_instance_2.get_instance_value()
mysubclass_instance.get_instance_value()

MyClass.set_instance_value(myclass_instance_1, 4)    # - - - instance methods can also be invoked by
class name, if supplied with an instance of that class - - -
MyClass.set_instance_value(myclass_instance_2, 5)
MySubclass.set_instance_value(mysubclass_instance, 6)

MyClass.get_instance_value(myclass_instance_1)
MyClass.get_instance_value(myclass_instance_2)
MySubclass.get_instance_value(mysubclass_instance)

# *** *** illustrate staticmethod invocation and operation

myclass_instance_1.set_static_value(1)           # - - - staticmethods can be invoked by class instance
myclass_instance_2.set_static_value(2)
mysubclass_instance.set_static_value(3)

myclass_instance_1.get_static_value()
myclass_instance_2.get_static_value()
mysubclass_instance.get_static_value()

MyClass.set_static_value(1)                      # - - - staticmethods are more commonly invoked drectly,
by class name; no instance is required - - -
MySubclass.set_static_value(2)
```

```
MyClass.get_static_value()
MySubclass.get_static_value()

# *** *** illustrate classmethod invocation and operation

myclass_instance_1.set_classattr_foo(1)         # - - - classmethods can be invoked by class instance
myclass_instance_2.set_classattr_foo(2)
mysubclass_instance.set_classattr_foo(3)

myclass_instance_1.get_classattr_foo()
myclass_instance_2.get_classattr_foo()
mysubclass_instance.get_classattr_foo()

MyClass.set_classattr_foo(1)                   # - - - classmethods are more commonly invoked directly,
by class name; no instance is required - - -
MySubclass.set_classattr_foo(2)

MyClass.get_classattr_foo()
MySubclass.get_classattr_foo()

############# end of examples #############################################
```

```
----------------------------------------------------------------------
*.  Data initialization, @staticmethod and @classmethod decorators
----------------------------------------------------------------------
```

In the previous example, calling one of MyClass's show methods before calling its corresponding set method will result in an AttributeError exception, caused by an attempt to access an attribute that has not yet been defined. These "use before define" errors can be prevented by initializing class and instance attributes when the class and its instances are first created.

Python provides different mechanisms for initializing class and instance attributes. Python supports the use of assignment statements in of class definitions to initialize static and class values. Python's default object constructor, __init__, is the standard Python mechanism for initializing instance data.

Python also supports a shortcut for defining static and classmethods. This shortcut involves the use of a second kind of functional, known as a **decorator**, for wrapping static and classmethods. A decorator is a construct of the form

```
  @something
  ... definition of object foo ...
```

that acts as a shorthand for

```
  .... definition of object foo ...
  foo = something(foo)
```

```
############# start of examples #############################################

# *** *** construct a class with instance, static, and classmethods

class MyClass:
  #
  static_value = 'initial static value'
  #
  @staticmethod
  def set_static_value(newv):   MyClass.static_value = newv
  @staticmethod
  def get_static_value():       return MyClass.static_value
  #
  foo = 'initial value in MyClass'
  #
  @classmethod
```

```
  def set_classattr_foo(thisclass, value):   thisclass.foo = value
  @classmethod
  def get_classattr_foo(thisclass):          return thisclass.foo
  #
  def __init__(self, initial_instance_value=None):
    self.instance_value = initial_instance_value
  #
  def set_instance_value(self, v):   self.instance_value = v
  def get_instance_value(self):      return self.instance_value

class MySubclass(MyClass):
  foo = 'initial value in MySubclass'

# confirm that the initializations succeeded

myclass_instance_1 = MyClass()
myclass_instance_2 = MyClass('initial instance value')
mysubclass_instance = MySubclass()

myclass_instance_1.get_instance_value()
myclass_instance_2.get_instance_value()
mysubclass_instance.get_instance_value()

myclass_instance_1.get_static_value()
myclass_instance_2.get_static_value()
mysubclass_instance.get_static_value()

myclass_instance_1.get_classattr_foo()
myclass_instance_2.get_classattr_foo()
mysubclass_instance.get_classattr_foo()

############# end of examples #############################################


-----------------------------------------
*.  Method invocation within a class
-----------------------------------------

    The examples included showed how to call methods that a class contains from
    outside of that class's scope.  In order to call a method from inside the
    class's scope, the method must be prefixed with either
    -.  a class name, or
    -.  self, for instance methods that act on the current object's data


############# start of examples #############################################

class MyClass:
  static_value = 'initial static value'
  #
  @staticmethod
  def set_static_value(newv):   MyClass.static_value = newv
  @staticmethod
  def get_static_value():      return MyClass.static_value
  #
  # --- reset static_value from **kwds, if present
  # --- initialize instance_value from **kwds, if present; otherwise, set to None
  #
  def __init__(self, **kwds):
    try:     MyClass.set_static_value( kwds[ 'static_value' ])
    except:  pass
    self.set_instance_value( kwds.get( 'instance_value', None ) )      # - - - the most common way of
calling an instance method from within class scope - - -
  #
  def add_to_instance_value(self, v):                                        # - - - using class name
with explicit 'self' argument also works - - -
    MyClass.set_instance_value(self, MyClass.get_instance_value(self) + v)    # - - - equivalent to
self.set_instance_value(self.get_instance_value() + v)
                                                                             # - - - or just
self.instance_value += v
```

```
  #
  def set_instance_value(self, v):     self.instance_value = v
  def get_instance_value(self):        return self.instance_value

# confirm that the initialization logic works as claimed

myclass_instance_1 = MyClass()
myclass_instance_1.get_static_value()
myclass_instance_1.get_instance_value()

myclass_instance_2 = MyClass(static_value = 'updated static value', instance_value = 'initial instance
value')
myclass_instance_2.get_static_value()
myclass_instance_2.get_instance_value()

myclass_instance_2.add_to_instance_value( ', now with additional content')
myclass_instance_2.get_instance_value()

############# end of examples ############################################
```

```
-------------------------------------------------------------------------------
*.  Superclass-subclass interaction:  exploiting and avoiding virtualization
-------------------------------------------------------------------------------
```

    In an OO language, the following sequence of actions creates a situation
    where the language's run-time system must choose between two methods with
    the same name:

    -. an object obj_A of type class_A invokes a method m_1 defined in
        one of A's superclasses, superclass_A
    -. superA method m_1, in turn, invokes a second method m_2 that has
        two definitions:
        -. a first in superclass_A, superclass_A.m_2
        -. a second in class_A, class_A.m_2

    If the language, when confronted with this choice, chooses class_A.m_2
    over superclass_A.m_2, method m_2 is said to be *virtual*. The term,
    "virtual", is (to me) an unfortunate and confusing name for a policy that
    "simply" says,
       "when an ambiguity arises with regard to choosing between two methods
        with a common name, favor the original object's point of view
        over the superclass's."

    Treating methods as virtual by default is regarded as a best practice in
    OO language design. In Python, virtualization is a natural consequence of
    the use of "self" to qualify method names. To avoid virtualization, invoke
    an instance method with an explicit reference to its container class.

```
############# start of examples ##########################################

# *** *** illustrating calling conventions for virtual and non-virtual method calls *** ***

class MyClass:
  def __init__(self, iv_1, iv_2):
    self.set_instance_value_1( iv_1 )              # - - - a virtual method call - - -
    MyClass.set_instance_value_2( self, iv_2 )     # - - - a non-virtual method call - - -
  #
  def set_instance_value_1(self, v):    self.instance_value_1 = 'set from MyClass: ' + v
  def get_instance_value_1(self):       return self.instance_value_1
  #
  def set_instance_value_2(self, v):    self.instance_value_2 = 'set from MyClass: ' + v
  def get_instance_value_2(self):       return self.instance_value_2

class MySubclass(MyClass):
  def set_instance_value_1(self, v):    self.instance_value_1 = 'set from MySubclass: ' + v
  def get_instance_value_1(self):       return self.instance_value_1
  #
  def set_instance_value_2(self, v):    self.instance_value_2 = 'set from MySubclass: ' + v
```

```
  def get_instance_value_2(self):       return self.instance_value_2

# confirm that the initialization logic works as claimed

mysubclass_instance = MySubclass( 'first', 'second' )
mysubclass_instance.get_instance_value_1()
mysubclass_instance.get_instance_value_2()

############# end of examples #############################################
```

```
---------------------------------------------------------------
*.  Superclass-subclass interaction:  multiple inheritance
----------------------------------------------------------------
```

Python supports **multiple inheritance**: the ability of a class to acquire attributes from two or more superclasses. One common use of multiple inheritance involves the creation of *mixins*: classes that contain logic -- typically, complex logic -- that could potentially find use in two or more client classes. Mixins support a "once-and-only-once" approach to logic design: a key strategy for enhancing a code's maintainability.

To specify that a class inherits from two or more classes, list that class's superclasses in its header, in order of priority. Classes that appear to the left in list take priority over classes that appear to the right in case of name conflicts (if any) involving superclass attributes.

A class's primary superclass is recorded in that class's __class__.__base__ attribute. The classes from which a class inherits can be identified by invoking *any* (!) class's __class__.mro() method (short for "method resolution order") with the class of interest as its argument. Similarly, a class's subclasses can be identified by invoking *any* (!) class's __class__.__subclasses__() method on the class of interest.

```
############# start of examples ###########################################

# *** *** illustrating multiple inheritance and the resolution of inheritance conflicts *** ***

class MyMainClass:
  def __init__(self, v):    self.set_instance_value_1(v)
  #
  def set_instance_value_1(self, v):    self.instance_value_1 = 'set from MyClass: ' + v
  def get_instance_value_1(self):       return self.instance_value_1

class MyMixinClass_1:
  def set_instance_value_2(self, v):    self.instance_value_2 = 'set from MyMixinClass_1: ' + v
  def get_instance_value_2(self):       return self.instance_value_2
  #
  def set_instance_value_4(self, v):    self.instance_value_4 = 'set from MyMixinClass_1: ' + v
  def get_instance_value_4(self):       return self.instance_value_4

class MyMixinClass_2:
  def set_instance_value_3(self, v):    self.instance_value_3 = 'set from MyMixinClass_2: ' + v
  def get_instance_value_3(self):       return self.instance_value_3
  #
  def set_instance_value_4(self, v):    self.instance_value_4 = 'set from MyMixinClass_2: ' + v
  def get_instance_value_4(self):       return self.instance_value_4

class MySubclass(MyMainClass, MyMixinClass_1, MyMixinClass_2):  pass

# *** *** confirming the class hierarchy *** ***

# - - -  base classes - - -

MyMainClass.__class__.__base__            # primary superclass
object.__class__.mro(MyMainClass)[1:]     # all superclasses

MyMixinClass_1.__class__.__base__         # primary superclass
object.__class__.mro(MyMixinClass_1)[1:]  # all superclasses
```

```
MyMixinClass_2.__class__.__base__         # primary superclass
object.__class__.mro(MyMixinClass_2)[1:]  # all superclasses

MySubclass.__class__.__base__             # primary superclass
object.__class__.mro(MySubclass)[1:]      # all superclasses

# - - - subclasses - - -

object.__class__.__subclasses__(MyMainClass)

object.__class__.__subclasses__(MyMixinClass_1)

object.__class__.__subclasses__(MyMixinClass_2)

object.__class__.__subclasses__(MySubclass)

# *** *** showing the action of the final, mixin-based class class methods *** ***

my_subclass_instance = MySubclass('one')
my_subclass_instance.set_instance_value_2('two')
my_subclass_instance.set_instance_value_3('three')
my_subclass_instance.set_instance_value_4('four')

my_subclass_instance.get_instance_value_1()   # from MyClass
my_subclass_instance.get_instance_value_2()   # from MyMixinClass_1
my_subclass_instance.get_instance_value_3()   # from MyMixinClass_2
my_subclass_instance.get_instance_value_4()   # from MyMixinClass_1, which shadows MyMixinClass_2

############# end of examples ############################################
```

----
*. Shorthands for managing instance data: properties

```
    The property() functional, a Python builtin, allows an identifier to serve
    as a stand-in for a combination of a getter, a setter, a deleter, and/or
    a docstring setter.  This method can be invoked in one of three ways:

    -.  As a function with keywords.  A call like
            foo = property(fget=get_foo, fset=set_foo, fdel=del_foo, fdoc=foo_docstring)
          in an instance x of a class allows client codes to use expressions like
            -.  print(x.foo)  in lieu of  print(x.get_foo())
            -.  x.foo = 3     in lieu of  x.set_foo(3)
            -.  del x.foo     in lieu of  x.del_foo()
          and  x.foo.__doc__ as a synonym for docstring.
        Note that the keyword version of this idiom allows for the definition
        of a property that lacks any of these four methods.

    -.  As a function with positional parameters.  Calls to
        -.  foo = property(get_foo)
        -.  foo = property(get_foo, set_foo)
        -.  foo = property(get_foo, set_foo, del_foo)
        -.  foo = property(get_foo, set_foo, del_foo, foo_docstring)
        define foo as a property with a get method; with get and set methods;
        with get, set, and del methods; and with get, set, and del methods and
        the foo_docstring docstring, respectively.  To define a property that
        lacks one or more of these methods, that parameter can be passed as
        "None":  e.g.,
        -.  foo = property(get_foo, None, None, foo_docstring)
            defines a read-only property with a docstring.

     -.  Using property decorators.  Here,
         -.  an initial decorator, @property, is used to specify that a function
             -- say, foo-- foo is a getter for a property named foo:
                @property
                def foo(self):
                    """ a docstring for foo, if one is wanted, goes here """
                    ....
```

```
         -.  subsequent decorators with the property name prepended to the
             decorator are then used to specify the property's setter and/or
             deleter, if required.  The functions being decorated must also
             have the property's name:  e.g.,
                @foo.setter
                def foo(self, value):
                    ....
                @foo.deleter(self):
                    ....

       When defining a property, use a name for that property that differs from
       the names of all of a class's instance variables.  Using the same name
       for a property and an instance variable "confuses" the Python
       interpreter:  attempting to access such a name causes Python to repeatedly
       access the name as a property rather than the name as an attribute of
       self, leading ultimately to stack overflow and a program crash.

############# start of examples #############################################

# *** *** three examples that show the use of a property to provide different *** ***
# *** *** views of what, essentially, is the same datum: a circle's radius    *** ***

# **********************************************************
# *** example 1:  using the "foo = property(foo)" idiom ***
# **********************************************************

class Circle(object):
  pi = 3.14159
  #
  def __init__(self, **kwargs):
    print(set(kwargs.keys()))
    if len(set(['radius', 'diameter', 'circumference', 'area']) & set(kwargs.keys())) != 1:
       raise KeyError("constructor must be called with exactly one keyword from 'radius', 'diameter',
'circumference', or 'area'")
    try:  self.radius = kwargs['radius']
    except KeyError:
      try:  self.diameter = kwargs['diameter']
      except KeyError:
        try:  self.circumference = kwargs['circumference']
        except KeyError:
          self.area = kwargs['area']
  #
  def getradius(self):
    if not 'r' in dir(self):  raise UnboundLocalError("radius undefined")
    return self.r
  def setradius(self, r):  self.r = r
  def delradius(self):
    if 'r' in dir(self):   del self.r
  radius = property(getradius, setradius, delradius, 'circle radius')
  #
  def getdiameter(self):
    try:       return self.radius * 2
    except:       raise UnboundLocalError("diameter undefined")
  def setdiameter(self, d):  self.radius = d / 2
  def deldiameter(self):     del self.radius
  diameter = property(getdiameter, setdiameter, deldiameter, 'circle diameter')
  #
  def getcircumference(self):
    try:       return 2 * Circle.pi * self.radius
    except:       raise UnboundLocalError("circumference undefined")
  def setcircumference(self, c):  self.radius = c / (2 * Circle.pi)
  def delcircumference(self):     del self.radius
  circumference = property(getcircumference, setcircumference, delcircumference, 'circle circumference')
  #
  def getarea(self):
    try:       return Circle.pi * self.radius * self.radius
    except:       raise UnboundLocalError("circumference undefined")
  def setarea(self, a):   self.radius = pow(a / Circle.pi, 0.5)
  def delarea(self):      del self.radius
  area = property(getarea, setarea, delarea, 'circle area')
```

```
c = Circle(radius=3)
c.radius, c.diameter, c.circumference, c.area
c.area = 3
c.radius, c.diameter, c.circumference, c.area
del c.circumference
c.radius, c.diameter, c.circumference, c.area


# ***********************************
# *** example 2:  using decorators  ***
# ***********************************

class Circle(object):
  pi = 3.14159
  #
  def __init__(self, **kwargs):
    if len(set(['radius', 'diameter', 'circumference', 'area']) & set(kwargs.keys())) != 1:
       raise KeyError("constructor must be called with exactly one keyword from 'radius', 'diameter',
'circumference', or 'area'")
    try:  self.radius = kwargs['radius']
    except KeyError:
      try:  self.diameter = kwargs['diameter']
      except KeyError:
        try:  self.circumference = kwargs['circumference']
        except KeyError:
          self.area = kwargs['area']
  #
  @property
  def radius(self):
    """circle radius"""
    if not 'r' in dir(self):  raise UnboundLocalError("radius undefined")
    return self.r
  @radius.setter
  def radius(self, r):   self.r = r
  @radius.deleter
  def radius(self):
    if 'r' in dir(self):  del self.r
  #
  @property
  def diameter(self):
    """circle diameter"""
    try:       return self.radius * 2
    except:       raise UnboundLocalError("diameter undefined")
  @diameter.setter
  def diameter(self, d):  self.radius = d / 2
  @diameter.deleter
  def diameter(self):     del self.radius
  #
  @property
  def circumference(self):
    """circle circumference"""
    try:       return 2 * Circle.pi * self.radius
    except:       raise UnboundLocalError("circumference undefined")
  @circumference.setter
  def circumference(self, c):  self.radius = c / (2 * Circle.pi)
  @circumference.deleter
  def circumference(self):     del self.radius
  #
  @property
  def area(self):
    """circle area"""
    try:       return Circle.pi * self.radius * self.radius
    except:       raise UnboundLocalError("circumference undefined")
  @area.setter
  def area(self, a):   self.radius = pow(a / Circle.pi, 0.5)
  @area.deleter
  def area(self):      del self.radius

c = Circle(radius=3)
```

```
c.radius, c.diameter, c.circumference, c.area
c.area = 3
c.radius, c.diameter, c.circumference, c.area
del c.area
c.radius, c.diameter, c.circumference, c.area

############# end of examples #############################################


----------------------------------------------------------------------
*.  Superclass-subclass interaction:  accessing "shadowed" methods
----------------------------------------------------------------------

    When a method in a class C "shadows" a method m of the same name in one of
    that class's superclasses SuperC, the superclass method of the same name
    can be invoked in one of two ways:

    -.   using the syntax super().m( .... ), where "..." is short for the
         method's remaining parameters, **provided that** SuperC is the
         first class in object.__class__.mro(C)[1:] with a method m

    -.   using the syntax SuperC.m( self, ... ), where "..." is short for the
         method's remaining parameters, when the more flexible "super()" can't
         be used:  e.g., in multiple inheritance hierarchies with multiple
         superclasses with __init__ methods

    This need to call a superclass method with the same name as a subclass
    method arises commonly for class constructors, where it is used to delegate
    the work of initializing attributes defined in superclasses to the
    superclasses that define them.


############# start of examples #############################################

# ********************************************************************************
# *** *** accessing superclass methods with the same name as subclass methods *** ***
# ********************************************************************************

# *** *** case 1:  single inheritance *** ***

# --- --- case 1a:  user-defined class as superclass --- ---

class MyClass:
  def __init__(self, **kwds ):
    MyClass.set_instance_value_1( self, **kwds )
  #
  def set_instance_value_1(self, **kwds):
    self.instance_value_1 = ('value_1, as set by MyClass', kwds.get('value_1', None))
  def get_instance_values(self):
    return (self.instance_value_1,)

class MySubclass(MyClass):
  def __init__(self, **kwds ):
    super().__init__( **kwds )
    self.set_instance_value_2( **kwds )
  #
  def set_instance_value_2(self, **kwds ):
    self.instance_value_2 = ('value_2, as set by MySubclass', kwds.get('value_2', None))
  def get_instance_values(self):
    return super().get_instance_values() + (self.instance_value_2,)

# - - - - - - confirming the methods' operation - - - - - -

my_subclass_instance_1 = MySubclass()
my_subclass_instance_1.get_instance_values()

my_subclass_instance_2 = MySubclass(value_1=1)
my_subclass_instance_2.get_instance_values()

my_subclass_instance_3 = MySubclass(value_2=2)
```

```
my_subclass_instance_3.get_instance_values()

my_subclass_instance_4 = MySubclass(value_1=1, value_2=2)
my_subclass_instance_4.get_instance_values()

# --- --- case 1b:  built-in class as superclass --- ---

class MyList(list):
  def __init__(self, l, owner ):
    super().__init__( l )
    self.owner = owner
  #
  def get_owner(self):
    return self.owner

# - - - - - - confirming the methods' operation - - - - - -

my_list = MyList( [1, 2, 3, 4], 'Phil Pfeiffer' )
my_list                # the instance behaves like an ordinary list instance...
my_list.get_owner()    # ...up to its having an additional, owner attribute


# *** *** case 2:  multiple inheritance, duplicate attribute names across classes *** ***

class MyMainClass:
  def __init__(self, v):   self.set_instance_value_1( v )
  #
  def set_instance_value_1(self, v):    self.instance_value_1 = 'set from MyMainClass: ' + v
  def get_instance_value_1(self):       return self.instance_value_1

class MyMixinClass:
  def __init__(self, v2 ):
    self.set_instance_value_2( v2 )
  #
  def set_instance_value_2(self, v):    self.instance_value_2 = 'set from MyMixinClass: ' + v
  def get_instance_value_2(self):       return self.instance_value_2

# - - - simpler here, I think, to name the classes then to refer to super(),
# - - - which would change if the order of inheritance is varied

class MySubclass(MyMainClass, MyMixinClass):
  def __init__(self, v1, v2 ):
    MyMainClass.__init__( self, v1 )
    MyMixinClass.__init__( self, v2 )

my_subclass_instance = MySubclass('one', 'two')
my_subclass_instance.get_instance_value_1()   # from MyClass
my_subclass_instance.get_instance_value_2()   # from MyMixinClass

############# end of examples #############################################
```

----

*. Customizing relational operators: in particular, __eq__

Each of Python's relational operators implements its comparison by invoking a special attribute associated with its left-hand operand:

-. < invokes __lt__
-. <= invokes __le__
-. == invokes __eq__
-. != invokes __ne__
-. >= invokes __ge__
-. > invokes __gt__

These operations can be tailored to a class's semantics by redefining them in a class's definition. Python requires these attributes to be defined as functions of two arguments:

-. self, the current object's state

```
    -.  other, the state of the object with which to compare the current
        object: i.e., the relational operator's right-hand operand.

    Ordering comparisons between objects are not always appropriate.  By
    convention, a comparison between incomparable objects should return
    NotImplemented.  Python converts NotImplemented to False in the absence of
    a special check for NotImplemented.

    Every user-defined class **should** be assessed to determine appropriate
    definitions for __eq__ and __ne__.  This is of particular concern for
    immediate subclasses of object, since object.__eq__ and __ne__ use id()
    to test for equality:  i.e.,

         object_a == object_b    iff
             object_a and object_b reference the same (memory) object

    Using id() to test for equality is inappropriate when objects should be
    compared by behavior rather than identity.  One common strategy for
    crafting behavior-centric definitions is to check for equal type and
    state:  i.e.,

         object_a == object_b  iff
              object_a and object_b are of a common type and
              object_a and object_b have the same state

    One surprising implementation wrinkle is that Python apparently tests
    for a == b by checking *both* objects' __eq__ attributes:  i.e.,

        a == b iff  a.__eq__(b) and b.__eq__(a)

    This policy, which differentiates Python from (e.g.) Ruby, which omits
    the "right to left" check, assures the symmetry of ==.  It does not,
    however, ensure that __eq__ is transitive:  see below for examples.

#############  start of examples  ###########################################

# The following code shows three examples of classes with overloaded relational
# operators.  All three examples are based on Python's built-in list class.
# The first two examples show cases where __eq__ seems to work well.  The third
# shows how definitions of __eq__ can lead to trouble in the presence of
# subclassing.

# *** ***  example 1:  list with "existential" comparison operators  *** ***
#
# The following definition of relational operations is one that XSLT uses for
# comparing lists.  Essentially, it holds that a <compare> b is true iff
# a[i] <compare> b[j] for some a[i] in a and b[j] in b

class Elist(list):
  def __lt__(self, other):
    return NotImplemented if not isinstance(other, Elist) else any([a <  b for a in self for b in other])
  def __le__(self, other):
    return NotImplemented if not isinstance(other, Elist) else any([a <= b for a in self for b in other])
  def __eq__(self, other):
    return isinstance(other, Elist) and any([a == b for a in self for b in other])
  def __ne__(self, other):
    return not isinstance(other, Elist) or any([a != b for a in self for b in other])
  def __ge__(self, other):
    return NotImplemented if not isinstance(other, Elist) else any([a >= b for a in self for b in other])
  def __gt__(self, other):
    return NotImplemented if not isinstance(other, Elist) else any([a >  b for a in self for b in other])

list0 = Elist([])
list1 = Elist([0, 1, 2, 3])
list2 = Elist([3, 4, 5, 6])
list3 = Elist([6, 7, 8, 9])

list0 < []
list0 == []
list0 != []
```

```
(list0 < list0, list0 <= list0, list0 == list0, list0 != list0, list0 >= list0, list0 > list0)
(list0 < list1, list0 <= list1, list0 == list1, list0 != list1, list0 >= list1, list0 > list1)

(list1 < list1, list1 <= list1, list1 == list1, list1 != list1, list1 >= list1, list1 > list1)

(list1 < list2, list1 <= list2, list1 == list2, list1 != list2, list1 >= list2, list1 > list2)
(list2 < list1, list2 <= list1, list2 == list1, list2 != list1, list2 >= list1, list2 > list1)

(list2 < list3, list2 <= list3, list2 == list3, list2 != list3, list2 >= list3, list2 > list3)
(list3 < list2, list3 <= list2, list3 == list2, list3 != list2, list3 >= list2, list3 > list2)

(list1 < list3, list1 <= list3, list1 == list3, list1 != list3, list1 >= list3, list1 > list3)
(list3 < list1, list3 <= list1, list3 == list1, list3 != list1, list3 >= list1, list3 > list1)


# *** ***  example 2:  list with an additional "owner" attribute  *** ***
#
# -.  use list's built-in ordering relations to compare lists
# -.  require the same owners for list equality

class MyList(list):
  def __init__(self, l, owner ):
    super().__init__( l )
    self.owner = owner
  #
  def __eq__(self, other):
    return isinstance(other, MyList) and super().__eq__(other) and self.owner == other.owner
  def __ne__(self, other):
    return not isinstance(other, MyList) or super().__ne__(other) or  self.owner != other.owner

mylist_123_phil = MyList([1, 2, 3], 'Phil')
mylist_456_phil = MyList([4, 5, 6], 'Phil')
mylist_123_bob  = MyList([1, 2, 3], 'Bob')

(mylist_123_phil == mylist_123_phil, mylist_123_phil != mylist_123_phil)
(mylist_123_phil == mylist_456_phil, mylist_123_phil != mylist_456_phil)
(mylist_123_phil == mylist_123_bob,  mylist_123_phil != mylist_123_bob)


#############  end of examples  #############################################
```

+++
8. OO Programming in Python: Design Considerations
+++

----
*. Data hiding

Like other OO languages, Python supports encapsulation. An object B that is contained in an object A must, by default, be referenced "through" A, using expressions like A.B. While it is possible to make an external module's contents directly visible to another module, this also requires a "reaching through" the external module's "exterior", using syntax like

```
from external_module import entity
```

Unlike most OO languages with which I'm familiar-- and this includes C++, C#, Java, Eiffel, and Ruby-- Python does not support data hiding as a language feature. Python lacks the equivalent of declarations like "public", "private" and "protected", which are common to other OO languages. With the exception of a few built-in immutable classes and constants, all Python identifers are readable and updateable, as are the objects they reference. I've seen this lack of checking justified on the grounds of efficiency and flexibility: i.e.,

-. Enforcing data hiding in the absence of static typechecking would impose a significant overhead on every data access, most of which would involve useless checks.

-. Programmers should have the freedom to break data hiding whenever the dictates of a problem warrants this.

Python, however, does support one weak mechanism for data hiding, as well as a second, somewhat stronger mechanism for protecting attributes against access. The weak mechanism, name mangling, is invoked automatically for attributes whose names start with a double underscore. Python manages these attributes by "silently"

-. assigning them new names and
-. converting all references to these attributes from within their container objects to these new names

This transforming of those attributes' "real" names effectively invalidates all external references to the "real" names. This transformation, however, is easily circumvented, as shown in the examples below.

The other approach to name mangling involves redefining __getattribute__ and __setattr__, special attributes for retrieving and updating an object's attributes. With a few exceptions involving special attributes like __hash__, access to an object's attributes can be controlled by recoding the following special attributes:

-. __getattribute__, which dereferences identifiers in contexts where where their values are being read
-. __setattr__, which dereferences identifiers in contexts where their values are being updated

However, as shown below, techniques similar to those required for this recoding can be used to bypass the recoded __getattribute__ and __setattr__.

The __getattribute__ and __setattr__ attributes can also be recoded so as to respond to requests for attributes that the class does not explicitly define. This "trick" is comparable the use of Ruby's :methodNotImplemented method to respond to undefined method calls: an idiom that the Ruby community refers to-- somewhat confusingly-- as "virtual methods".

```
#############  start of examples  ###########################################

# *** example 1:  'weak' data hiding using data mangling

class MyClass:
  def __init__(self, v):  self.__value = v
  def get_value(self):    return self.__value

item = MyClass(1)

item.__value           # this fails...
item.__value = 4       # ...as does this...
item.get_value()       # ... as demonstrated by this method call

dir(item)              # examining item, however, reveals the subterfuge

item._MyClass__value       # this ploy succeeds...
item._MyClass__value = 4   # ... as does this...
item.get_value()           # ... as demonstrated by this method call


# *** example 2:  'strong' data hiding using __getattribute__ and __setattr__

# - - -  the overloading of __getattribute__ and _setattr_ in the class below
# - - -  this disables external references to value v through item

class MyClass(object):
  def __init__(self, v, w):
    super().__setattr__('v', v)     # trying to access 'v' with MyClass's (redefined) _getattribute__ and
__setattr__ will fail
    self.w = w
  #
```

```
  def get_v(self):
    return super().__getattribute__('v')
  def set_v(self, v):
    return super().__setattr__('v', v)
  #
  def __getattribute__(self, attr_name):
    if attr_name == 'v':  raise LookupError("v is private -- can't be read")
    #
    # what follows-- as well as "return object.__getattribute__(attr_name)"--
    # are the standard fallback actions for "all other attributes"
    #
    return super().__getattribute__(attr_name)
  def __setattr__(self, attr_name, v):
    if attr_name == 'v':  raise LookupError("v is private -- can't be written")
    #
    # what follows-- as well as "return object.__setattr_(a, v)"--
    # are the standard fallback actions for "all other attributes"
    #
    return super().__setattr__(attr_name, v)

item = MyClass(1, 2)
item.w          # w is directly accessible ...
item.w += 2     # ... and updatable
item.w

item.v          # v, however, is neither directly accessible ...
item.v += 2     # ... nor updateable ...

item.set_v(item.get_v() + 2)   # .. though it can be manipulated through the class's interface
item.get_v()

# - - -  note, however, that data hiding can still be defeated externally, using object

object.__getattribute__(item, 'v')


# *** example 3:  overloading __getattribute__ to support Ruby-style "virtual methods"

class DeptOffices:
  all_dept_offices = \
     dict((name, (office, 'Nicks')) if not isinstance(office, tuple) else (name, office)
          for (name, office) in
            {('Don', 'Bailes'): 459, ('Gene', 'Bailey'): 477, ('Marty', 'Barrett'): 463,
             ('Sonya', 'Batchelder'): 464,  ('Vijay', 'Bhuse'): 483, ('Sam', 'Burke'): (107,
'Gilbreath'),
             ('Terry', 'Countermine'): 465, ('Jeremiah', 'Dangler'): 481, ('Todd', 'Franklin'): 461,
             ('Stephen', 'Hendrix'): 476, ('Mohammed', 'Hoque'): 486, ('Jay', 'Jarman'): 474,
             ('Selim', 'Kalayci'): 485, ('Mike', 'Lehrfeld'): 470, ('Robert', 'Nielsen'): 475,
             ('Carolyn', 'Novak'): 457, ('Adam', 'Ogle'): 487, ('Phil', 'Pfeiffer'): 467,
             ('Bill', 'Pine'): 460, ('Tony', 'Pittarese'): 484 , ('Kellie', 'Price'): 468,
             ('Jeff', 'Roach'): 473, ('David', 'Robinson'): ((6, 'B'), 'Wilson-Wallis'),
             ('Suzanne', 'Smith'): 471, ('David', 'Tarnoff'): 469, ('Chris', 'Wallace'): 478}.items())
  #
  def __init__(self, building=None):
    self.this_building = building
  #
  def __getattribute__(self, attr_name):
    if any(attr_name in person_name for person_name in DeptOffices.all_dept_offices.keys()):
      return [(name, (room, building)) for (name, (room, building)) in DeptOffices.all_dept_offices.items
() \
                   if attr_name in name and self.this_building in [None, building]]
    else:
      return super().__getattribute__(attr_name)

all_offices=DeptOffices()
all_offices.Phil
all_offices.Don
all_offices.Lehrfeld

gilbreath_offices=DeptOffices('Gilbreath')
```

```
gilbreath_offices.Phil
gilbreath_offices.Sam

gilbreath_offices.all_dept_offices

all_offices.foo

#############  end of examples  #############################################
```

```
--------------------------
*.  Dependency inversion
--------------------------

    The term "dependency inversion" dates to a time when objects supplanted
    procedures as the primary basis for software design.  The term refers to
    the structuring of a family of modules that serve the same essential
    function as codes with a common interface.  This "interchangeable parts"
    approach to program design was developed as an alternative to "run-time
    type inference" (RTTI):  a design strategy that

    -.  treats related modules that have different preconditions for operation
        as different  entities, and
    -.  selects which module to call at run-time via a series of tests that
        assess the current execution context.

    RTTI is now in disfavor, since it hinders program evolution. In RTTI-based
    designs, adding modules to or removing modules from a given family of
    modules can force changes to *all* codes that access *any* of these
    modules.  Dependency inversion, by contrast, localizes such changes to
    the modules proper, so long as those modules' common interfaces can be
    kept contstant.

    The first two examples below show the use of RTTI and dependency inversion,
    respectively, to implement a query-driven program action.  The second
    example's implementation simulates the standard idiom for dependency
    inversion in strongly typed languages like C++, C#, and Java:

    -.  The example treats all binary query objects as subclasses of a common,
        "binary response" base class.
    -.  The example uses this "binary response" class to verify that its
        "get_time" method has been supplied with a binary query object:
        a type check that "classic' languages do at compile-time instead.

    Using a base class to represent a family of related objects is a standard
    practice in languages with compile-time typing.  Algorithms for compile-
    time type checking, as a rule, require the type of every identifier to
    be declared explicitly and held fixed.  These representative base classes,
    as a rule, are constructed as **abstract** base classes:  classes that
    can be subclassed but cannot be used to instantiate objects directly.  In
    keeping with this practice, the example's base class has been defined as an
    instance of a Python library module, ABCMeta, that prevents a class C from
    instantiating new objects of type C.

    Python's native type-checking algorithm is far looser than C++'s, C#'s, or
    Java's.  Python's lack of mandatory type declarations, together with its
    support for "on-the-fly" code generation via eval() and exec(), make
    compile-time typing problematic at best and impossible at worst.  Python,
    rather, implements a form of run-time type checking, which assesses
    whether the statements it executes and the objects that appear in those
    statements reference

    -.  attributes that can be read and/or written, according to the sense of
        the contexts in which they appear, and, in the case of objects,
    -.  possess the well-known attributes, like __dir__ (for calls to dir())
        and __call__ (for calls to identifiers) that these contexts require.

    This approach to typing is referred to as "latent typing" or "duck typing",
    as in, "if it walks like a duck and talks like a duck, it's a duck".
    This approach to typing is favored by a minimalist school of software
```

design, including authorities like Russ Olsen (_Eloquent Ruby_) who argue for achieving software quality by writing
-. concise programs with clearly named variables
-. that are supported by carefully crafted regression test suites.

Authorities on dynamically typed languages like Olsen routinely criticize hard-coded type checks as constructs that clutter programs: i.e., that attempt to address problems that would be better addressed by careful attention to naming and regression testing. While they encourage the use of inheritance for eliminating duplicate code, they see it as useless for supporting the defining of types.

As someone who sees value in type checking, I'm not sure if I agree with this "minimalist" position. Still, I've included a third, Olsen-style implementation of the second example as a basis for comparison. Note that the code in this third example fails to catch the error created by the use of a method name, queryOption, to designate a method that returns four possible values instead of just two.

```
#############  start of examples  ###########################################

# *************************
# *** *** example 1 *** ***
# *************************
#
#  using RTTI to structure a code that solicits
#  -.  an "option 1 or 2" answer from a user in one of several languages, then
#  -.  launches a command in response to this response

import time

def QueryInEnglish():
  while True:
    option = input('Please enter 1 for 24 hour format, 2 for AM/PM format: ')
    if option in ["1", "2"]: return option

def QueryInFrench():
  while True:
    option = input('S\'il vous pla\EEt entrer 1 pour le format de 24 heures, 2 pour le format AM / PM: ')
    if option in ["1", "2"]: return option

def QueryInSpanish():
  while True:
    option = input('Por favor, introduzca 1 para el formato de 24 horas, 2 para el formato AM / PM: ')
    if option in ["1", "2"]: return option

def get_time(language):
  optstrings = {'1': "%H:%M", '2': "%I:%M %p"}
  if language=="EN":
    return time.strftime(optstrings[QueryInEnglish()])
  elif language=="FR":
    return time.strftime(optstrings[QueryInFrench()])
  elif language=="ES":
    return time.strftime(optstrings[QueryInSpanish()])
  else:
    return "language not supported: {0}".format(language)

get_time("EN")
1

get_time("FR")
2

get_time("ES")
3
1

get_time("IT")
```

```
# ************************
# *** *** example 2 *** ***
# ************************
#
#  using dependency inversion to structure a code that solicits
#  -.  an "option 1 or 2" answer from a user in one of several languages, then
#  -.  launches a command in response to this response
#
#  the code encapsulates queries as classes with an abstract base class,
#  and features a type check for the get_time() routine parameter

import time

# metaclasses are classes that control the instantiation of classes
# for more on metaclasses, see the resources described at the outset of this document

from abc import ABCMeta

class TwoOptionQuery(metaclass=ABCMeta):
  @classmethod
  def getOption(clas):
    while True:
      option = input(clas.querystring)
      if option in ["1", "2"]: return option

class QueryInEnglish(TwoOptionQuery):
  querystring = 'Please enter 1 for 24 hour format, 2 for AM/PM format: '

class QueryInFrench(TwoOptionQuery):
  querystring = 'S\'il vous pla\EEt entrer 1 pour le format de 24 heures, 2 pour le format AM / PM: '

class QueryInSpanish(TwoOptionQuery):
  querystring = 'Por favor, introduzca 1 para el formato de 24 horas, 2 para el formato AM / PM: '

class QueryInItalian:    # a four-option query that supports optional time zones
  @staticmethod
  def getOption():
    while True:
      option = input('Inserisci 1 per il formato 24 ore, 2 per il formato AM / PM, e 3 o 4 di questi
formati con fusi orari, rispettivamente:: ')
      if option in ["1", "2", "3", "4"]: return option

# note the use of __name__ to display the problem class's name in what follows
# note also the explicit check for subclass-class relationship
# this check would be automatic in compiled OO languages --
#   if desired, it must be explicit in interpreted languages
#
def get_time(querier):
  assert issubclass(querier, TwoOptionQuery), "querier ({0}) must be a subclass of TwoOptionQuery".format
(querier.__name__)
  return time.strftime({'1': "%H:%M", '2': "%I:%M %p"}[querier.getOption()])

get_time(QueryInEnglish)
1

get_time(QueryInFrench)
2

get_time(QueryInSpanish)
3
1

get_time(QueryInItalian)   # generates appropriate assertion on failure, due to class mismatch

# note that the check can be bypassed
#
time.strftime({'1': "%H:%M", '2': "%I:%M %p"}[QueryInItalian.getOption()]


# ************************
```

```
# *** *** example 3 *** ***
# *************************
#
#  example 3, simplified as per standard "duck typing" practice.
#  -.  the TwoOptionQuery superclass is retained, because it provides common implementation logic
#      for its child classes.
#  -.  the ABCMeta designation has been discarded, however, as as the assertion in get_time

class TwoOptionQuery:
  @classmethod
  def getOption(clas):
    while True:
      option = input(clas.querystring)
      if option in ["1", "2"]: return option

class QueryInEnglish(TwoOptionQuery):
  querystring = 'Please enter 1 for 24 hour format, 2 for AM/PM format: '

class QueryInFrench(TwoOptionQuery):
  querystring = 'S\'il vous pla\EEt entrer 1 pour le format de 24 heures, 2 pour le format AM / PM: '

class QueryInSpanish(TwoOptionQuery):
  querystring = 'Por favor, introduzca 1 para el formato de 24 horas, 2 para el formato AM / PM: '

class QueryInItalian:    # a four-option query that supports optional time zones
  @staticmethod
  def getOption():
    while True:
      option = input('Inserisci 1 per il formato 24 ore, 2 per il formato AM / PM, e 3 o 4 di questi
formati con fusi orari, rispettivamente:: ')
      if option in ["1", "2", "3", "4"]: return option

# note the use of __name__ to display the problem class's name in what follows
#
def get_time(querier):
  assert issubclass(querier, TwoOptionQuery), "querier ({0}) must be a subclass of TwoOptionQuery".format
(querier.__name__)
  return time.strftime({'1': "%H:%M", '2': "%I:%M %p"}[querier.getOption()])

get_time(QueryInEnglish)
1

get_time(QueryInFrench)
2

get_time(QueryInSpanish)
3
1

get_time(QueryInItalian)   # fails without being identified as a type error
4


#############  end of examples  #############################################
```

In addition to supporting a mechanism for defining abstract base classes, the Python library also provides decorators that cause a class's methods and properties to be treated as abstract methods and properties: i.e., objects that the class's subclasses are required to overload. See Section 28.7 of the Python Library Document, the section on Python's abc ("abstract base class") module, for details.

----
*. Docstrings

A docstring is a string that's referenced by an object's __doc__ attribute. By convention, docstrings document an object's purpose and provide test cases for that object's execution. The Python interpreter automatically

```
    binds a string literal that appears as the first statement of a function or
    class to that object's __doc__ attribute.

    Docstrings have a special significance for the Python library's doctest
    module, which
    -.  searches docstrings for substrings that are formatted as docstring-
        like test sessions, then
    -.  executes those commands to confirm that they return the specified
        results and/or have the desired effects.

    The examples below use a doctest module function, run_docstring_examples,
    to invoke doctest() on a single function.  The Python library manual's
    documentation on doctest gives further examples that show how to run
    doctest on entire modules-- i.e., .py files-- from the command line as
    well as from Python codes:  a mode of use that the manual says is much
    more common in practice.

#############  start of examples  ###########################################

def powers_of_2():
  """generate successive powers of 2, starting with 2^0
  """
  next_result = 1
  while True:
    yield next_result
    next_result *= 2

def print_first_n_powers_of_2(n):
  """  print the first n powers of 2, starting with 2^0

  routine prints the first n powers of 2 for a user-supplied integer n,
  starting with 2^0 and ending with 2^n-1.

  >>> print_first_n_powers_of_2(0.5)
  Traceback (most recent call last):
     ...
  TypeError: 'float' object cannot be interpreted as an integer
  >>> print_first_n_powers_of_2(-1)
  >>> print_first_n_powers_of_2(0)
  >>> print_first_n_powers_of_2(1)
  1
  >>> print_first_n_powers_of_2(5)
  1
  2
  4
  8
  16
  """
  p2 = powers_of_2()                      # obtain a copy of the generator function for local use
  for i in range(0,n):  print(next(p2))

# Python transforms the initial strings into docstrings
#
print(powers_of_2.__doc__)
print(print_first_n_powers_of_2.__doc__)

# Use doctest.run_docstring_examples to run print_first_n_powers_of_2's test cases, twice:
# -.  In a second mode that simply shows failed tests (default)
# -.  In a first mode that shows all test cases and their results as the cases execute
#
# More on docstring execution:
# -.  Test cases are signaled with initial ">>>" prompt strings.
# -.  Expected results are given after the commands to execute.
# -.  "Traceback" results are treated specially:
#     ..., with the ELLIPSIS option enabled, causes doctest to ignore Traceback details
#     when checking expected results.

import doctest

doctest.run_docstring_examples(print_first_n_powers_of_2, None, optionflags=doctest.ELLIPSIS)
```

```
doctest.run_docstring_examples(print_first_n_powers_of_2, None, optionflags=doctest.ELLIPSIS,
verbose=True)

#############  end of examples  #############################################

------------------
*.  Serialization
------------------

    Serialization is the saving of an object's state in a way that allows that
    object to be reinitialized at a later time.  Some common reasons for
    persisting objects include

    -.  The desire to dispatch part of a computation to another host,
        in order to use parallelism to make that computation run faster.
    -.  The desire to examine a computation's intermediate states, in order to
        -.  verify that program's correctness or
        -.  debug that program's execution
    -.  The desire to allow a computation to be restarted in the event of
        system failure:  a particular concern for computations that can take
        days to finish.
    -.  The desire to use the output of a first computation as the input to a
        second.

    Various third-party packages have been developed for serializing Python
    object content in relational databases.  These packages are commonly
    to as object-relational mappers, or ORMs for short.  The most widely used
    and well known ORM for Python is probably SQLAlchemy (www.sqlalchemy.org).
    SQL Alchemy supports a variety of popular back-end databases, including
    MySQL, PostgreSQL, and SQLite.  Its website includes extensive how-to
    documentation, including (e.g.) code for saving recursive objects in
    a back-end database.

    The Python distribution proper provides two lighter-weight mechanisms for
    serializing object content.  The one, pickling, is supported by the Python
    library's pickle module.  Pickling, which converts an object's state into
    a byte stream, is described in detail in the Python library.

    While ORMs and pickling seem useful, the only serialization mechanism that
    I've worked with to date is Python's repr() builtin.  Python classes, by
    convention, often include a special method, __repr__, that outputs a
    class instance's content in a way that allows that instance to be
    recreated, using eval().  One advantage of repr() methods for debugging is
    is they dump an object's contents in text, which makes a program's state
    easier to review.

    Serialization has its limitations, in that some types of Python objects
    resist serialization.  Opaque data types like code, for example, can neither
    be effectively pickled nor repr-ized, as shown below.  These limitations,
    however, are not ones that I've had to work around in my Python coding--
    at least, not to date.

#############  start of examples  ###########################################

# *** *** showing the repr-ization of a class instance and the subsequent
# *** *** reconstruction of an equivalent class instance
#
# Key points:
#
# -. The class is defined in a way that localizes all variable data that
#    a class instance depends on to that instance.  The class, for example,
#    does not define class-scope data that varies over time.  None of the
#    class's methods, moreover, reference global variables whose state must
#    be saved to reconstruct the instance's execution context.
#
# -. The class is defined with an __init__ method that exposes all of its
#    internal state, in the sense of allowing a constructor to initialize
#    all internal data directly.  Doing so allows for the definition of a
#    __repr__ method that outputs a constructor expression.
```

```
#
# -. The class's __repr__ method uses the :r qualifier to reprize the
#    individual components of the class's state.  This idiom is common
#    for repr methods.
#
# -. The class's __eq__ method has been defined to test for equivalence
#    based on equal state.  This definition is essential for using ==
#    to validate the definition of __repr__, as shown below

class MyClass:
  def __init__(self, iv_1, iv_2):  self.iv_1, self.iv_2 = iv_1, iv_2
  #
  def __repr__(self):  return "{0}({1!r},{2!r})".format(self.__class__.__name__, self.iv_1, self.iv_2)
  #
  def __eq__(self, other):  return isinstance(other, MyClass) and self.iv_1 == other.iv_1 and self.iv_2 
== other.iv_2
  def __ne__(self, other):  return not(self.__eq__(other))

x = MyClass([1, 2], {'three':3})

repr(x)   # showing how a repr-ized instance looks

eval(repr(x)) == x     # must evaluate to True if __repr__ has been defined correctly


#  illustrating the point that code objects resist repr-ization
#
f = lambda: 3
g = lambda: 3
repr(f)
repr(g)   # f and g are functionally equivalent, but have different repr's
f == g    # they are not, moreover, equal under Python's definition of equality

y = MyClass(1, f)
z = MyClass(1, g)
repr(y)
repr(z)    # similarly, y and z are equivalent, but have different repr's
y == z     # they also fail to test as equal

eval(repr(y))   # fails, because repr(f) isn't a valid Python expression

#############  end of examples  #############################################
```

----
*. Nested classes

Python, as noted earlier, supports the nesting of objects within objects, including functions within functions, as shown earlier, and classes within classes, as shown below. In this example, the Card class is nested inside the CardDeck class to express the notion that a Card only exists in the deck that contains it

```
#############  start of examples  ###########################################

class CardValues(object):
    def __init__(self, values = ['ace', 'king', 'queen', 'jack', '10', '9', '8', '7', '6', '5', '4', '3', 
'2']):
        self.vals = values
    def __len__(self):
        return len(self.vals)
    def __getitem__(self, i):
        return self.vals[i]
    def __eq__(self, other):
        return isinstance(other, CardValues) and len(self) == len(other) and all([self[i] == other[i] for 
i in range(0, len(self))])
    def values(self):  return self.vals
    def outranks(self, v1, v2):
        assert v1 in self.vals, "value ({0}) not in values ({1})".format(v1, self.vals)
```

```
        assert v2 in self.vals, "value ({0}) not in values ({1})".format(v2, self.vals)
        return self.vals.index(v1) < self.vals.index(v2)

class CardSuits(object):
    def __init__(self, suits = ['spades', 'hearts', 'diamonds', 'clubs']):
        self.suit_names = suits
    def __len__(self):
        return len(self.suit_names)
    def __getitem__(self, i):
        return self.suit_names[i]
    def __eq__(self, other):
        return isinstance(other, CardSuits) and len(self) == len(other) and all([self[i] == other[i] for
i in range(0, len(self))])
    def suits(self):  return self.suit_names
    def outranks(self, s1, s2):
        assert s1 in self.suit_names, "suit ({0}) not in suits ({1})".format(s1, self.suit_names)
        assert s2 in self.suit_names, "suit ({0}) not in suits ({1})".format(s2, self.suit_names)
        return self.suit_names.index(s1) < self.suit_names.index(s2)

class CardDeck(CardValues, CardSuits):
    class Card(object):
        def __init__(self, deck, suit, value):
            self.deck, self.suit, self.value = deck, suit, value
        def comparable(self, other):
            return isinstance(other, CardDeck.Card) and self.deck.suits() == other.deck.suits() and
self.deck.values() == other.deck.values()
        def __eq__(self, other):
            return self.comparable(other) and self.suit == other.suit and self.value == other.value
        def __gt__(self, other):
            return self.comparable(other) and deck.outranks(self, other)
        def __lt__(self, other):
            return self.comparable(other) and deck.outranks(other, self)
    def __init__(self, values=None, suits=None):
        if values == None:  CardValues.__init__(self)
        else:               CardValues.__init__(self, values)
        if suits == None:   CardSuits.__init__(self)
        else:               CardSuits.__init__(self, suits)
    def isSuit(self, suit):    return suit in self.suits()
    def isValue(self, value):  return value in self.values()
    def getCard(self, suit, value):
        assert self.isSuit(suit),    "suit  ({0}) missing from deck's suits  ({1})".format(suit,
self.suits())
        assert self.isValue(value),  "value ({0}) missing from deck's values ({1})".format(value,
self.values())
        return self.Card(self, suit, value)       # can also be CardDeck.Card
    def __eq__(self, other):
        return isinstance(other, CardDeck) and self.suits() == other.suits() and self.values() ==
other.values()
    def outranks(self, card1, card2):
        assert isinstance(card1, CardDeck.Card),  "card ({0}) missing from deck".format(card1)
        assert isinstance(card2, CardDeck.Card),  "card ({0}) missing from deck".format(card2)
        return card1.suit == card2.suit and CardValues.outranks(self, card1.value, card2.value)

# showing some operations on cards...
#
deck = CardDeck()
card1 = deck.getCard('spades', 'ace')
card2 = deck.getCard('spades', '2')
card3 = deck.getCard('clubs', 'ace')
card1 > card2
card1 > card3

############  end of examples  ############################################
```

----
*. Iterators

A class's __iter__ method determines the sequence of values that a class
instance returns when it's included in a loop.

```
    One of two standard idioms for using __iter__ is to define __iter__ as a
    self-contained generator for returning the desired sequence.  The other,
    more common idiom treats __iter__ as an initializer for a second special
    attribute, __next__.  In this second idiom,

    -.  __iter__ is defined as a two-part method that
        -.  initializes any state that's needed to support the iteration, then
        -.  returns self, as a way of handing off the computation to __next__.
    -.  __next__ then uses this internal state to return each element of
        the required sequence in succession

############  start of examples  ###########################################

# *** *** example 1:  showing the use of __iter__ as a self-contained generator
#
# the example uses the earlier fib function, recast as a class

class Fib:
  def __init__(self, val_count = float('inf')):
    self.val_count, self.prev_2, self.prev_1 = val_count, 0, 1
  #
  def __iter__(self):
    if self.val_count < 1:  raise StopIteration
    yield self.prev_2
    self.val_count -= 1
    if self.val_count < 2:  raise StopIteration
    yield self.prev_1
    self.val_count -= 1
    while self.val_count > 0:
      self.val_count -= 1
      yield self.prev_2 + self.prev_1
      self.prev_2, self.prev_1 = self.prev_1, self.prev_2 + self.prev_1

fibgen = Fib(10)

[i for i in fibgen]   # works once ...
[i for i in fibgen]   # ... but not twice, since val_count is never reset


# *** *** example 2:  showing the use of __iter__ in combination with __next__

class Fib:
  def __init__(self, val_count = float('inf')):
    self.initial_val_count = val_count
  #
  def __iter__(self):
    self.val_count, self.result_queue = self.initial_val_count, [0, 1]
    return self
  #
  def __next__(self):
    if self.val_count <= 0:  raise StopIteration
    self.val_count -= 1
    self.next_result = self.result_queue[0]
    self.result_queue = [self.result_queue[1], self.result_queue[0] + self.result_queue[1]]
    return self.next_result

fibgen = Fib(10)

[i for i in fibgen]   # works once ...
[i for i in fibgen]   # ... and a second time, since __iter__ resets val_count

############  end of examples  #############################################


+++++++++++++++++++++++++++++++++++++++++++++++++++++
9.  Modules
+++++++++++++++++++++++++++++++++++++++++++++++++++++

--------------------
```

*. Introduction

Previous exercises have shown the use of Python's import statement to augment the current Python environment with references to objects defined in the Python standard library. So far, two variants of the import command have been shown:

```
-.   import m1          # imports module m1, enabling use of references to objects m1 exports,
                        #    using syntax like m1.foo
                        # also supports the importation of multiple modules:  e.g.,  import m1, m2, m3
-.   from m1 import o1  # imports object o1 from m1, enabling use of references to o1
                        # also supports the importation of multiple objects:  e.g.,  from m1 import o1, o2, o3
```

Python supports two further variants of the import command:

```
-.   from m1 import o1 as a1    # imports object o1 from m1, enabling referencing of o1 under the alias a1
-.   from m1 import *           # imports all objects that m1 exports
```

The first of these two variants should be self-explanatory. The second will be covered in more detail below,
in the course of discussing the following concepts:

-. changing the interpreter's import path
-. managing the Python import cache
-. managing name conflicts with the Python standard library
-. mechanisms for supporting command-line-based module execution, including
    -. __name__ == '__main__'
    -. sys.argv
    -. sys.stdout, sys.stderr, exit()
    -. sys.stdin
-. using __all__ to specify module exports
-. creating **packages**: i.e., modules that share a common directory with an __init__.py file

----
*. Changing Python's default import path

The Python interpreter supports two types of modules: built-ins, which are compiled into Python, and file-based
modules, which are accessed via the interpreter's resident file system. A file-based module foo will be found
in a file named "foo.py", in a directory specified by Python's sys.path identifier. This sys.path identifier
lists directories to search when importing modules in order of precedence, much as

-. a standard CLI's PATH variable names directories to search for executables and
-. Java's CLASSPATH variable names directories to search for Java class libraries.

By updating sys.path, you can change how Python resolves the first attempt to import a given module.

############# start of examples ##

>>> begin this series of examples by opening a new Python session, then executing the following commands:

```
import sys
print sys.path
```

>>> identify two directories that are currently missing from the list of directories in the list sys.mypath
>>> for the sake of this example, we'll assume these directories are subdirectores of your current directory,
named mydir1 and mydir2.

>>> continue by creating two files:
>>>>>> a first, mydir1/foo.py, that contains one function, as follows:

```
def foo():  print("Hello - I'm foo, version 1")
```

```
>>>>>> a second, mydir2/foo.py, that contains one function, as follows:

def foo():  print("Hello - I'm foo, version 2")


>>> next, run the following command.  It should fail.

import foo


>>> next, run the following commands:

sys.path += ['mydir1', 'mydir2']
sys.path
import foo
foo.foo()

>>> next, **while keeping this Python session open**, change the content of mydir1/foo.py to read

def foo():  print("Hello - I'm foo, updated version 1")

>>> then run the following commands in the Python session

import foo
foo.foo()

>>> finally, **while keeping this Python session open**, run the following commands

sys.path = sys.path[:-2] + ['mydir2', 'mydir1']
sys.path
foo.foo()  # this still produces unexpected results for reasons explained below

#############  end of examples  #############################################
```

----
*. Managing Python's protocols for module importation

The last two executions of foo.foo() in the previous series of examples produce unexpected results because of Python's implementation of the import statement. When Python first imports a module 'foo', it uses sys.path to locate 'foo'. Upon locating foo, Python caches two of foo's attributes in sys.modules:

-. the name of the module being imported, in a key in sys.modules
-. the location of the imported module, in the key's corresponding value

Subsequent attempts to reimport 'foo' then
-. either have no effect, if foo is present in the session, or
-. reference the module location recorded in sys.modules, independently of sys.path, otherwise.

This cacheing of an imported module's attributes speeds the process of importing new modules when those modules, in turn, reference previously imported modules-- a considerable time savings in the case of commonly referenced modules. Cacheing, however, creates a need to circumvent caching when revising a module over the course of a given session.

The following example shows two strategies for circumventing cacheing:
-. the imp.reload() function, which works when reimporting a module whose source file's path remains constant
-. manipulating sys.modules, which is needed when reimporting a module whose source file's path has changed

```
#############  start of examples  #############################################

>>> begin this series of examples by opening a new Python session, then executing the following commands:

import sys
print sys.path
```

```
>>> identify two directories that are currently missing from the list of directories in the list
sys.mypath
>>> for the sake of this example, we'll assume these directories are subdirectores of your current
directory,
named mydir1 and mydir2.

>>> continue by creating two files:
>>>>>> a first, mydir1/foo.py, that contains one function, as follows:

def foo():  print("Hello - I'm foo, version 1")

>>>>>> a second, mydir2/foo.py, that contains one function, as follows:

def foo():  print("Hello - I'm foo, version 2")


>>> next, run the following commands:

sys.path += ['mydir1', 'mydir2']
sys.path
import foo
foo.foo()

>>> next, **while keeping this Python session open**, change the content of mydir1/foo.py to read

def foo():  print("Hello - I'm foo, updated version 1")

>>> then run the following commands in the Python session

sys.path = sys.path[:-2] + ['mydir2', 'mydir1']
sys.path
import imp
imp.reload(foo)
foo.foo()

>>> finally, **while keeping this Python session open**, run the following commandssys.path = sys.path
[:-2] + ['mydir2', 'mydir1']
sys.path
del sys.modules['foo']
del foo
import foo
foo.foo()

############  end of examples  #############################################

---------------------------------------------------------------
*.  Managing name conflicts with the Python standard library
---------------------------------------------------------------

    Start a new Python session, and run a command like

       for i in sys.modules.items():  print(i)

    to see a list of modules that Python pre-loads when it begins execution.  If, for some obscure
    or misguided reason, you try to import a module with one of these names, Python will ignore the
    attempt to import that module, much as it ignored the attempt to access foo from mydir2 in the
    first set of exercises above.  If, for some obscure or misguided reason, you really want to import
    a module named (say) 'errno', 'os', or 'io', you can do so by first deleting the relevant key-value
    pair from sys.modules before importing your module.   I would, however, strongly advise against
    reusing any of the names assigned to these standard Python modules.


---------------------------------------------------------
*.  Making Python's modules aware of execution context
---------------------------------------------------------

    A Python module can be run as a program if that module is supplied as the first argument to
    the Python interpreter.  In addition to enabling command-line-interpreter- (CLI-) based module
    execution, Python provides features that allow a module to do the following:
```

```
    -.  determine whether it's been invoked from a command line
    -.  for CLI-based executions,
        -.  access the command line arguments with which it was called
        -.  return output to the session window, via sys.stdout or sys.stderr
        -.  accept input from a session window, via sys.stdin
        -.  return a result code to the CLI environment

#############  start of examples  #############################################

>>> begin this series of examples by creating a file, 'foo.py', with the following content

prev, this = 1, 1
print(prev)
print(this)
for i in range(3,11):
    prev, this = this, this+prev
    print(this)

>>> exit your editor, raise a command prompt, and execute 'python foo.py' (without quotes)
>>> to run this program

>>> now, update foo to read as follows:

if __name__ == '__main__':
    prev, this = 1, 1
    print(prev)
    print(this)
    for i in range(3,11):
        prev, this = this, this+prev
        print(this)
else:
    prev, this = 55, 34
    print(prev)
    print(this)
    for i in range(3,11):
        prev, this = this, -this+prev
        print(this)

>>> run this program twice:
>>>>>   once from a command prompt, as 'python foo.py'  (again, without quotes)
>>>>>   once from an interactive python session, by executing  'import foo'   (again, without quotes)

#############  end of examples  #############################################

-------------------------------------------------------------
*.  Accessing command-line arguments and execution context
-------------------------------------------------------------

    Python's sys module includes objects that act as references to resources in standard POSIX
    environments.  They include
    -.  sys.argv -
            the command line arguments that a command interpreter (CLI) passes to a program
            when it begins execution
    -.  sys.stdin, sys.stdout, sys.stderr -
            a program's standard input, output, and error streams, respectively

    The Python standard library also supports access to an executable's environment variables.
    This facility, however, is provided through the os library module variable os.environ.


#############  start of examples  #############################################

>>> Begin this series of examples by saving the following code to a file-- say, bar.py.
>>> This code also illustrates a use of Python's regular expression module--here, to check
>>> for strings that can be parsed as integers.

import sys, re

for (argno, arg) in enumerate(sys.argv):
```

```
    if not(re.match('[+-]?\d+$',arg)):
        print('arg {0} ({1}) is not of type int'.format(argno, arg), file=sys.stderr)

print("sum of integer arguments in argv is {0}".format(sum([int(i) for i in sys.argv if re.match('[+-]?\d+
$',i)])))


>>>  then, try executing it with command lines like the following:

    python bar.py 1 a 2 b 3 c
    python bar.py 1 a 2 b 3 c 1> nul     # use 1> /dev/null from a Unix environment;  "nul" is windows
    python bar.py 1 a 2 b 3 c 2> nul     # use 1> /dev/null from a Unix environment;  "nul" is windows


>>> Here's an example that shows the use of os.environ to recover all environment variables that
>>> start with user-specified strings


import sys, os, re

# find and print all environment variables that begin with strings specified by
# this program's command-line arguments  (excepting sys.argv[0], the program's name)
#
for arg in sys.argv[1:]:
    matching_environment_variables = [(k, v) for (k, v) in os.environ.items() if re.match(arg, k)]
    if len(matching_environment_variables) > 0:
        print("environment variables beginning with {0}:".format(arg))
        maxkeylength = max([len(k) for (k, v) in matching_environment_variables])
        m_e_v_sorted = sorted(matching_environment_variables, key=lambda v: v[0])
        for (k, v) in m_e_v_sorted:
            print("     {0}:{1}  {2}".format(k, ' '*(maxkeylength-len(k)), v))


>>> try saving it to a file-- say, bar.py-- and running it with a few different arguments:  e.g.,

    python bar.py A E G X
    python bar.py P

#############  end of examples  #############################################


--------------------------------------------------
*.  Using __all__ to control what modules export
--------------------------------------------------

    The variable __all__ specifies what identifiers are loaded into a namespace following
    the execution of
        from ...module-name... import *

#############  start of examples  #############################################

>>> Begin this series of examples by saving the following code to a file in your Python
>>> interpreter's import path-- say, bar.py.

__all__ = ['f', 'g']

def f(x):  return x

def g(x):  return h(x)+1

def h(x):  return 2*x


>>>  start an interactive Python session.  Try running the following commands.

from bar import *

f(3)

g(3)
```

```
h(3)

bar.h(3)

import bar

h(3)

bar.h(3)     # here, note that h is still accessible as bar.h, even though omitted from __all__

from bar import h

h(3)

#############  end of examples  #############################################


--------------------------------------------------
*.  Packages
--------------------------------------------------

    A package, roughly speaking, is
    -.  a set of modules in a common directory, together with
    -.  (a possibly empty) file, __init.py__, that
        -.  is executed when the module is loaded, and
        -.  appears to be useful for three purposes:
            -.  updating the application's runtime environment
            -.  making modules available to the "from module import *" statement,
                using the __all__ variable
            -.  adding object names to the current scope, using import statements

    The two hedges in that description are deliberate:
    -.  The phrase "roughly speaking" is used because a package's designer can use a package
        attribute named __path__ to extend a package's scope to multiple directories.  As the
        Python tutorial notes, however, this use of __path__ is infrequent.
    -.  The phrase "appears to be useful" is used because of the lack of good documentation
        on how __init__ can be used to initialize a module's codes.  After a fair amount of
        experimentation, I've discovered the following:
        -.  The __all__ feature works as described (e.g.) in Summerfield
        -.  __init__ is useful for relieving callers of the need to do a series of imports:
            e.g., for making a function available from a package module under a package-
            author-determined name.
        -.  __init__ appears to be useful for initializing other objects in a package's scope.
            Modules in the package, however, must explicitly import the names of those objects
            in order to access them.  This includes names of other package modules.
        -.  If there's a way of using __init__ to update global variables, I haven't found it.

#############  start of examples  #############################################

>>> Begin this series of examples by creating a subdirectory in the current directory-- say, mod--
>>> and creating the following three files in the mod subdirectory:

>>>>>  mod\__init__.py, as follows:

from mod import foo, bar

>>>>>  mod\foo.py, as follows:

from mod.bar import bar_fn
def foo_fn(x):  return bar_fn(x) + 1

>>>>>  mod\bar.py, as follows:

def bar_fn(x):  return x+7

# note that the base directory for cross-module references in packages appears to be
# the package directory's parent directory
#
# note also the use of dot notation to cross-reference modules in packages.  . and ..
```

```
# both appear to be supported with their usual meanings.

>>>> start an interactive Python interpreter.  Try the following commands:

import mod
dir()              # note that only the name "mod" is in the current workspace
foo_fn(3)          # should fail
foo.foo_fn(3)      # should fail
mod.foo.foo_fn(3)  # should succeed

>>>> ***keeping the current Python session active***, try changing "x+7" in bar.py to "x*7"
>>>> then, try the following commands

mod.foo.foo_fn(3)   # nothing changes

>>>> next, try the following to get the change to take

import imp
imp.reload(mod)
mod.foo.foo_fn(3)

>>>> next, try the following to get the change to take

imp.reload(mod.bar)
mod.foo.foo_fn(3)

>>>> next, try the following to get the change to take

imp.reload(mod.bar)
mod.foo.foo_fn(3)

>>>> from what I can tell, in order to get a change to a package to take in the course of a given session,
>>>> all modules that are involved in the package must be reloaded, along with the package proper.
>>>> as per my initial comment, I haven't found clear documentation to this effect in the Python
notes ... yet.

>>> finally, try the following

>>>>>>  add the following two lines to  __init__.py

from mod.foo import foo_fn
__all__ = ['foo_fn']

>>>>>> exit and restart Python
>>>>>> run the following commands

from mod import *
dir()
foo_fn(3)

#############  end of examples  #############################################



+++++++++++++++++++++++++++++++++++++++++++++++++++++
10.  Concluding examples
+++++++++++++++++++++++++++++++++++++++++++++++++++++

**********************************************************
   Retrieving a random percentage of a text file's lines
**********************************************************

import random

# strategy 1: load entire file into memory before processing it
#
def random_subfile( file_name, percent ):
  #
  # from Python library, 10.1.2. Itertools Recipes
  #
```

```
  def random_combination(iterable, r):
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.sample(range(n), r))
    return tuple(pool[i] for i in indices)
  #
  assert (0 < percent <= 100 ), 'percentage argument ({0}) not in range'.format(percent)
  #
  # reduce file to a vector of lines
  #
  file_handle = open( file_name )
  file_content = file_handle.read().splitlines()
  file_handle.close()
  #
  # then, return the specified percentage of lines as a vector
  #
  return [file_content[i] for i in random_combination( range(len(file_content)), len(file_content) *
percent // 100 )]

random_subfile('a tour of python.txt', 10)



*************************************
   Stupid-simple primes generation
*************************************

# pass #1: classic, very stupid algorithm for testing if x is prime:
# -.   divide x by all values greater than 1 and less than x; do any divide evenly?

def way_stupid_factor_checker(potential_prime):
  is_prime = True
  for potential_divisor in range(2, potential_prime):
    if potential_prime % potential_divisor == 0:
      print("{0} // {1} = {2}".format(potential_divisor, potential_prime, potential_prime //
potential_divisor))
      is_prime = False
  return is_prime

def do_prime_test(potential_prime, factor_checking_function):
   print("{0} is{1} prime".format(potential_prime, "n't" if not(factor_checking_function
(potential_prime)) else ''))

def do_way_stupid_prime_test(potential_prime):
   do_prime_test(potential_prime, way_stupid_factor_checker)

do_way_stupid_prime_test(2)
do_way_stupid_prime_test(97)
do_way_stupid_prime_test(323)
do_way_stupid_prime_test(324)


# pass #2:  eliminate loop

def way_stupid_factor_checker(potential_prime):
  return ["{0} // {1} = {2}".format(potential_prime, potential_divisor, potential_prime //
potential_divisor)
               for potential_divisor in range(2, potential_prime)
               if potential_prime % potential_divisor == 0
         ]

def do_prime_test(potential_prime, factor_checking_function):
   non_prime_cases = factor_checking_function(potential_prime)
   for case in non_prime_cases:  print(case)
   print("{0} is{1} prime".format(potential_prime, "n't" if len(non_prime_cases) else ''))

do_way_stupid_prime_test(2)
do_way_stupid_prime_test(97)
do_way_stupid_prime_test(323)
do_way_stupid_prime_test(324)
```

```
# pass #3:  shrink range
#
# we only have to test through ceil(sqrt(potential_prime+1)), since this implicitly checks all larger
values

from math import ceil, sqrt

# mathematicians have figured out how to do much better -- but this is a start

def stupid_factor_checker(potential_prime):
  return ["{0} // {1} = {2}".format(potential_prime, potential_divisor, potential_prime //
potential_divisor)
               for potential_divisor in range(2, ceil(sqrt(potential_prime+1)))
               if potential_prime % potential_divisor == 0
         ]

def do_stupid_prime_test(potential_prime):
   do_prime_test(potential_prime, stupid_factor_checker)

do_stupid_prime_test(2)
do_stupid_prime_test(97)
do_stupid_prime_test(323)
do_stupid_prime_test(324)


# pass #4:  eliminate verbosity - limit to testing for primes

def has_factors(potential_prime):
  return any([potential_prime % potential_divisor == 0 for potential_divisor in range(2, ceil(sqrt
(potential_prime+1)))])

has_factors(2)
has_factors(97)
has_factors(323)
has_factors(324)


# pass #5:  transform into a function that lists primes in a given range

primes_up_to = lambda value:  [potential_prime for potential_prime in range(2,value+1) if not(has_factors
(potential_prime))]

primes_up_to(2)
primes_up_to(97)
primes_up_to(323)
primes_up_to(324)

# -- making it more concise by folding dependent function in-line --

primes_up_to = lambda v:  [p for p in range(2,v+1) if not(any([p % potential_divisor == 0 for
potential_divisor in range(2, ceil(sqrt(p+1)))]))]

primes_up_to(2)
primes_up_to(97)
primes_up_to(323)
primes_up_to(324)

# -- making it more concise by eliminating "not" --

primes_up_to = lambda v:  [p for p in range(2,v+1) if all([p % d != 0 for d in range(2, ceil(sqrt(p
+1)))])]

primes_up_to(2)
primes_up_to(97)
primes_up_to(323)
primes_up_to(324)
```

```
**********************************************
   Python dispatcher example  (showing "eval")
**********************************************

def do_tasks(*task_list):
  for (task, parameters) in task_list:
    try:
      eval(task)(*parameters)
    except Exception as err:
      print("exception for task {0}: {1}".format(task, err))

def clean():
   print("I'm cleaning the environment")

def allocate_cores(num_cores):
   print("I'm allocating {0} cores".format(num_cores))

def setenv_with_dict(env_values):
  for (env_var, env_value) in env_values.items():
    print("I'm setting {0} to {1}".format(env_var, env_value))

def setenv_with_pair_list(*env_values):
  for (env_var, env_value) in env_values:
    print("I'm setting {0} to {1}".format(env_var, env_value))

task_list =\
  [('clean',[]), \
   ('allocate_cores',[4]), \
   ('setenv_with_dict', [{'a': 'alpha', 'b': 'beta', 'g':'gamma'}]),\
   ('setenv_with_pair_list', [('d', 'delta'), ('e', 'epsilon'), ('p', 'pi')]),\
   ('unsupported_action', [])
]

do_tasks(*task_list)


*********************
   Game of Life
*********************

*.  Conway's game of life, as a CLI-based Python script  (with contributions from Adam Ogle)
    (see http://en.wikipedia.org/wiki/Conway's_Game_of_Life for an explanation)

    Imagine how long this program would be if it were written in C++, C#, or Java.

############  start of examples  ##########################################

import random, time

# set the size of the simulation
world_size = 50

# generate the initial world
world = [[not random.randint(0,7) for j in range(world_size)] for i in range(world_size)]

# returns sum of cells in a neighborhood of a cell [i][j] of a 2-D array, "a"
neighborhood_sum = lambda a, i, j: sum(a[(i+k) % len(a)][(j+l) % len(a[i])] for k in range(-1,2) for l in
range(-1,2)) - a[i][j]

# start the simulation
while True:
  # update the world
  world = [[(lambda count: count == 3 or world[i][j] and count == 2)(neighborhood_sum(world,i,j)) for j
in range(len(world[i]))] for i in range(len(world))]
  #
  # print the world
  for i in world: print(*["X" if j else " " for j in i], sep="")
  print("-"*len(world))
  #
  time.sleep(0.3)
```

```
#############  end of examples  #############################################
```