CMU MSP 36602 Python Review & File Handling April 2019

1) **Assignment**:
   a. use "="
   b. names can have "_" but not "."
   c. separated_word_style is preferred
   d. items in a returned list or tuple can be assigned individually using code like:

      a, _, b = myReturnThree(myArg)

2) **Weird copy behavior**:
   a. `x2 = x1` does not make a copy of the data, only a new pointer to the data
   b. subsequent `x1[i] = j` will also change x2
   c. but x1 = x3 will not change x2
   d. Fix with x2 = x1.copy(). You may need deepcopy() for nested objects.

3) **Variable types**:
   a. None, bool, int, float, complex, str
   b. list, tuple, set, dict
      `dict.get()` vs. `dict[]`: can check for `is None` for the former vs. error for the latter
   c. range, list comprehension, generator

4) **Indexing** is zero based; `x[2:10:3]` is R's `x[c(3, 6, 9)]`

5) **Indenting** (recommended: 4 spaces) defines code blocks

6) "**for loops**" use the format `for spam in eggs:`
   a. `continue` and `break` alter loop completion
   b. `while expression:` is also used

7) **List comprehensions** are key for vectorization
   a. `[f(x) for x in iterable]` returns an vector with altered versions of the values in the iterable
   b. `[f(i) for i in range(n)]` has length n
   c. `[f(x) for x in iterable if g(x)==h]` may be shorter than the iterable
   d. `[f(a, b, c) for (a, b, c) in zip(x, y, z)]` is like R's `apply(cbind(x,y,z), 1, f)`
   e. `[f(i, x) for (i, x) in enumerate(y)]` gives access to the index and the value
   f. `[f(x, y) for x in iter1 for y in iter2]` is like a double "for" loop
   g. `list(map(f, iterable))` is like `[f(x) for x in iterable]`

h. `list(map(f, iter1, iter2))` is like `[f(a, b) for (a, b) in zip(iter1, iter2)]`

8) **Expressions** use "and", "or", and "not"
9) **Operators** include e.g., `5 in range(5)` (which is False)
10) **Built-in functions** include
    a. `dir()`, `vars()`, `help()`
    b. `len()`, `type()`
    c. `all()`, `any()`
    d. `abs()`, `pow()`, `min()`, `max()`, `sum()`, `round()`, `divmod()`
    e. `list()`, `set()`, `dict()`, `bool()`, `int()`, `float()`, `chr()`, `ord()`
    f. `print()`, `repr()`
    g. `sorted()`, `reversed()`
    h. `range()`, `map()`: `list(map(abs, range(-2, 2))) -> [2, 1, 0, 1]`
    i. `zip()`, `enumerate()`: the latter is like `zip(range(length(x)), x)`.
    j. `open()` opens files for reading or writing
11) "is None" and "is not None" are used for **testing for None**
12) E.g., `isinstance(spam, int)`, is used for other **type testing**
13) E.g., `lambda x, y: (x, y, x**y)` makes an anonymous single-statement function
14) Errors are reported (and generated) as **Exceptions**, with **try handling**
15) **Functions** are defined with, e.g., `def spam(eggs, swallows="laden"):`
    a. `""" Comment """` as the first line(s) of a function is the preferred documentation mechanism
    b. Use of `return returnVal` is recommended, otherwise `None` is returned
16) **Import** loads functions, classes (and sometimes data) that are not built-in
    a. import math … x = math.nan
    b. import numpy as np … x = np.ndarray(range(5))
    c. from math import pi …. x = pi/2.0
17) Key statistical packages include **numpy** and **pandas**

18) File handling in Python

a. `os` and `dir` modules

b. `open(file, mode="r", buffering=-1, encoding=None,`
   `errors=None, newline=None, closed=True)`
   returns a file object (a.k.a. file handle or stream).

   The `file` argument is a string pointing to a file.

   The mode argument is a string with one or two characters in the form “ab” where “a” is one of:

   i. “`r`” for read
   ii. “`w`” for write (erasing the file if it already exists)
   iii. “`x`” for create (writes, but returns an error if the file already exists)
   iv. “a” for append (writes at the end of the file or creates a new one)
   v. “`w+`” (only with “b”) for binary random access (erasing existing data)
   vi. “`r+`” (only with “b”) for binary random access (preserving existing data)

   and “b” is either “`t`” for text (the default) or “b” for binary.

   The default for `buffering` implies buffering with a buffer size that is chosen automatically (usually 4KB or 8KB), except for interactive text files. You can also specify 0 to switch buffering off (only allowed in binary mode), 1 to select line buffering (only usable in text mode), and an integer > 1 to indicate the size of a fixed-size buffer in bytes.

   The `encoding` argument is only pertinent in text mode, and if encoding is not specified when in text mode, the encoding used is platform dependent: `locale.getpreferredencoding(False)` is called to get the current locale encoding.

   `errors` (text mode only) is an optional string that specifies how encoding errors are to be handled. Use “`strict`” (the default) to raise a ValueError exception if there is an encoding error or “`ignore`” to ignore errors.

   On input, if `newline` is `None` (the default), universal newlines mode is enabled, which means that lines in the input can end in '\n', '\r', or '\r\n', and these are translated into '\n'. If the argument is '', then line endings are untranslated. Other values can be used to specify custom end-of-line characters. On output, if `newline` is `None` (the default) then '\n' is written as the system default line separator (os.linesep), but if the argument is set to `''` or '\n', then no translation is done.

   If `closefd` is `False`, then the underlying file descriptor will be kept open when the file is closed. This does not work when a file name is given and must be True in that case.

   The `open()` function raises `IOError` on failure.

c. Some confusing similar functions:

Ref: https://stackoverflow.com/questions/15039528/what-is-the-difference-between-os-open-and-os-fdopen-in-python

i. Built-in open() takes a file name and returns a new Python file object. This is what you need in the majority of cases.

ii. os.open() takes a file name and returns a new file descriptor. This file descriptor can be passed to other low-level functions, such as os.read() and os.write(), or to os.fdopen(), as described below. You will probably never need this.

iii. os.fdopen() takes an existing file descriptor — typically produced by Unix system calls such as pipe() or dup(), and builds a Python file object around it. Effectively it converts a file descriptor to a full file object, which is useful when interfacing with C code or with APIs that only create low-level file descriptors.

d. Using `with`

Ref: https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files

It is good practice to use the `with` keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point. Using `with` is also much shorter than writing equivalent try-finally blocks:

```
with open('options.txt') as f:
    read_data = f.read() # can have several lines of code here
f.closed  # True (don’t include this in your code!)
```

If you’re not using the `with` keyword, then you should call f.close() to close the file and immediately free up any system resources used by it. If you don’t explicitly close a file, Python’s garbage collector will eventually destroy the object and close the open file for you, but the file may stay open for a while. Another risk is that different Python implementations will do this clean-up at different times.
After a file object is closed, either by a `with` statement or by calling f.close(), attempts to use the file object will automatically fail.

e. Controlling exceptions when working with files

```
try:
    with open("options.txt") as f:
       options = f.readlines()
except IOError as error:
    print('cannot read options – using defaults')
    options = my_defaults
```

f. **Binary storage using the `struct` module.**

Remember that a file is just a string of bytes.

In binary storage, we need to think about how the bytes are represented/interpreted:

```
"\N{GREEK SMALL LETTER ALPHA}+1"  # 'α+1'
"\N{GREEK SMALL LETTER ALPHA}+1".encode("utf-8")  # b'\xce\xb1+1'
[hex(x) for x in _]  # ['0xce', '0xb1', '0x2b', '0x31']
```

8 byte integers as bytes:
`hex(2048+512+128+32+8+1)  # '0xaa9'`
so the bytes are: `00 00 00 00 00 00 0a a9`
On a "little-endian" system this is `a9 0a 00 00 00 00 00 00`.

Real numbers are coded via IEEE754 in 8 bytes (little-endian or big-endian).

The Python `struct` model (https://docs.python.org/3.7/library/struct.html) defines data types (from the C perspective) that are converted to bytes via a "format string".

The main format codes are:

- s for string
- q for 8-byte integer
- d for double
- x for "padding"

The strings must be converted to bytes using a specific encoding and the number of characters is specified as a (base 10) number in front of the `s`.

As an example, `"d d 10s q d"` (or `"2d 10s q d"`) could be used to encode the storage of two doubles (confusingly called "float" in Python), a 10 character string (padded with null (0) bytes or truncated as needed), an integer, and a third double.

The format string can have a "prefix", and the default prefix is unsafe! An initial "<" specifies "little-endian" or ">" specifies big-endian. An explicit specification of big or little endian is ***required*** to prevents byte swapping when the read and write systems differ, and to prevent C-type alignment of integers and doubles via extra padding.

Use `struct.calcsize(myFormatString)` to see how much space something uses:

```
struct.calcsize("<16s q d")  # 32
struct.calcsize("<20s q d")  # 42
```

This is all used in one of three ways:

i. fully defined file specifications, e.g., get info A by going to B and reading C bytes

ii. storage of locations of variable-length info: at A find the address of info B

iii. record based storage: at (A+offset)*K find the K bytes for item A

g. Example code: See fileHandling.py