Journal for Lab3

Hongyan Yi (yih@oregonstate.edu)

1. What Programs I did?

Program 1: (5pts). Adapt the “is_numeric” function to acknowledge that float and int are not the ONLY numeric types! Check on Python.org to find at least two other numeric types, and change the “is_numeric” function to correctly recognize those as numeric types. In your scripts file, show that it works by passing at least one of the two other types.

Program 2: (5pts). Adapt the “assign_shape_params” function so it will test whether dim1 and dim2 are numeric and both positive numbers, returning to the user with a complaint if either one is not.

Program 3: (5pts). To the function “calc_shape_area,” add calculations of area to the other shapes that are defined in the “assign_shape_params” function.

Program 5: (10pts). Develop a function that scans a directory for files of a certain type (i.e. “.txt” or “.py” or “.shp”, etc.), sorts them according to a file characteristic (e.g. time stamp of creation, size of file in bytes, etc.) and writes a new text file with the names of all of the files that meet some criterion of that file characteristic (e.g. only new files, only files of a certain size, etc.).

Program 11: (10pts). Develop function to read a ‘.csv’ file with X, Y coordinates, and then use it to develop a program that will find the closest X, Y coordinate in a text file to a given target X, Y. coordinate. You can use any tool you’d like to read the ‘.csv’ file.

2. What knowledge I learned?

2.1 python numeric types

I try to test special numeric, still couldn't figure out why the yellow ones are not recognized by the function ?
http://www.tutorialspoint.com/python/python_numbers.htm

int	long	float	complex
10	51924361L	0.0	3.14j
100	-0x19323L	15.20	45.j
-786	0122L	-21.9	9.322e-36j
080	0xDEFABCECBDAECBFBAEL	32.3+e18	.876j
-0490	535633629843L	-90.	-.6545+0J
-0x260	-052318172735L	-32.54e100	3e+26J
0x69	-4721885298529L	70.2-E12	4.53e-7j

Literals that start with `0x` are hexadecimal integers. (base 16)

The number `0x6400` is `25600`.

```
6 * 16^3 + 4 * 16^2 = 25600
```

- **int (signed integers)**: They are often called just integers or ints, are positive or negative whole numbers with no decimal point.
- **long (long integers)**: Also called longs, they are integers of unlimited size, written like integers and followed by an uppercase or lowercase L.
- **float (floating point real values)** : Also called floats, they represent real numbers and are written with a decimal point dividing the integer and fractional parts. Floats may also be in scientific notation, with E or e indicating the power of 10 (2.5e2 = 2.5 x 10^2 = 250).
- **complex (complex numbers)** : are of the form a + bJ, where a and b are floats and J (or j) represents the square root of -1 (which is an imaginary number). The real part of the number is a, and the imaginary part is b. Complex numbers are not used much in Python programming.

2.2 python style

2.2.3 about space

If operators with different priorities are used, consider adding whitespace around the operators with the lowest priority(ies). Use your own judgment; however, never use more than one space, and always have the same amount of whitespace on both sides of a binary operator.

Yes:

```
i = i + 1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

No:

```
i=i+1
submitted +=1
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)
```

Avoid extraneous whitespace in the following situations:

Immediately **inside** parentheses, brackets or braces.

Yes:

```
spam(ham[1], {eggs: 2})
```

No:

```
spam( ham[ 1 ], { eggs: 2 } )
```

Immediately **before** a comma, semicolon, or colon:

Yes:

```
if x == 4: print x, y; x, y = y, x
```

No:

```
if x == 4 : print x , y ; x , y = y , x
```

However, in a slice the colon acts like a binary operator, and should have equal amounts on either side (treating it as the operator with the lowest priority). In an extended slice, both colons must have the same amount of spacing applied. Exception: when a slice parameter is omitted, the space is omitted.

Yes:

```
ham[1:9], ham[1:9:3], ham[:9:3], ham[1::3], ham[1:9:]
ham[lower:upper], ham[lower:upper:], ham[lower::step]
ham[lower+offset : upper+offset]
ham[: upper_fn(x) : step_fn(x)], ham[:: step_fn(x)]
ham[lower + offset : upper + offset]
```

No:

```
ham[lower + offset:upper + offset]
ham[1: 9], ham[1 :9], ham[1:9 :3]
ham[lower : : upper]
ham[ : upper]
```

2.3 return None VS return VS no return

On the actual behavior, there is no difference. They all return None and that's it. However, there is a time and place for all of these. The following instructions are basically how the different methods should be used (or at least how I was taught they should be used), but they are not absolute rules so you can mix them up if you feel necessary to.

2.3.1 Using return None.

This tells that the function is indeed meant to return a value for later use, and in this case it returns None. This value None can then be used elsewhere. return None is never used if there are no other possible return values from the function.

In the following example, we return person's mother if the person given is a human. If it's not a human, we return None since the "person" doesn't have a mother (let's suppose it's not an animal or so).

```
def get_mother(person):
    if is_human(person):
        return person.mother
    else:
        return None
```

2.3.2 Using return.

This is used for the same reason as break in loops. The return value doesn't matter and you only want to exit the whole function. It's extremely useful in some places, even though you don't need it that often.

We got 15 prisoners and we know one of them has a knife. We loop through each prisoner one by one to check if they have a knife. If we hit the person with a knife, we can just exit the function cause we know there's only one knife and no reason the check rest of the prisoners. If we don't find the prisoner with a knife, we raise an the alert. This could be done in many different ways and using return is probably not even the best way, but it's just an example to show how to use return for exiting a function.

```
def find_prisoner_with_knife(prisoners):
    for prisoner in prisoners:
        if "knife" in prisoner.items:
            prisoner.move_to_inquisition()
            return # no need to check rest of the prisoners nor raise an alert
    raise_alert()
```

Note: You should never do var = find_prisoner_with_knife(), since the return value is not meant to be caught.

2.3.3 Using no return at all.

This will also return None, but that value is not meant to be used or caught. It simply means that the function ended successfully. It's basically the same as return in void functions in languages such as C++ or Java.

In the following example, we set person's mother's name, and then the function exits after completing successfully.

```
def set_mother(person, mother):
    if is_human(person):
        person.mother = mother
```

Note: You should never do var = set_mother(my_person, my_mother), since the return value is not meant to be caught.

confused about return and return none in python, when return still print none

2.4 Tuple

make tuple to return idicator and value of a function is very useful.

```
if os.path.isfile(fn):  # os.path.isfile() returns True if fn really exists
    return True, fn     # return "true" and the filename
else:
    print("Error in concatenate_path_and_filename: File does not exist")

    return False, None    #if the file doesn't exist, return false.
```

2.5 handle delimiter for different operating system

we need to make sure that the path actually ends with the current operating system's path delimiter. On Windows machines, it's the \ slash, and on Macs and Unix/Linux it's a / slash. Fortunately, the "os" module keeps track of this for us!

2.5.1 handle separator by hand

```
if path[-1] != os.path.sep: #if the user did not put a delimiter at ↵
the end, then we try adding it
    try:
        path=path+os.path.sep   #use the operating system's separator
```

for example

```
import os
path = "X:\\class\\GIS 578\\Lab3\\week3"
path = path + os.path.sep
print path
X:\class\GIS 578\Lab3\week3\
print str(path)
X:\class\GIS 578\Lab3\week3\
```

2.5.2 handle separator by built-in

os.path.join module to take away having to deal with the delimiter ourselves

```
try:
    #this will handle the delimiter for us.
    fn = os.path.join(path, filename)
```

```
import os
path = "X:\\class\\GIS 578\\Lab3\\week3"
filename = "a.txt"
complete = os.path.join(path,filename)
print complete
X:\class\GIS 578\Lab3\week3\a.txt
```

2.6 try-except VS if-else

(1) hint from codebook

```
try:
    #this will handle the delimiter for us.
    fn = os.path.join(path, filename)

except IOError as e:
    print("Error in check filename:  Problem with either the path or the
    filename")
    print("I/O error {0}: {1}".format(e.errno, e.strerror))
    return False, None
except:
    print("Error in check filename:  Undetermined.")

    return False, None
```

(2) hint from website

http://stackoverflow.com/questions/1835756/using-try-vs-if-in-python

Is there a rationale to decide which one of `try` or `if` constructs to use, when testing variable to have a value?

For example, there is a function that returns either a list or doesn't return a value. I want to check result before processing it. Which of the following would be more preferable and why?

```
result = function();
if (result):
    for r in result:
        #process items
```

or

```
result = function();
try:
    for r in result:
        #process items
except TypeError:
    pass;
```

n your example (say that instead of returning a list or an empty string, the function were to return a list or `None`), if you expect that 99 % of the time `result` will actually contain something iterable, I'd use the `try/except` approach. It will be faster if exceptions really are exceptional.
If `result` is `None` more than 50 % of the time, then using `if` is probably better.
To support this with a few measurements:

```
>>> import timeit
>>> timeit.timeit(setup="a=1;b=1", stmt="a/b") # no error checking
0.06379691968322732
>>> timeit.timeit(setup="a=1;b=1", stmt="try:\n a/b\nexcept ZeroDivisionError:\n pass")
0.0829463709378615
>>> timeit.timeit(setup="a=1;b=1", stmt="if b!=0:\n a/b")
0.11940114974277094
```

```
>>> timeit.timeit(setup="a=1;b=0", stmt="try:\n a/b\nexcept ZeroDivisionError:\n pass")
0.5070195056614466
>>> timeit.timeit(setup="a=1;b=0", stmt="if b!=0:\n a/b")
0.051202772912802175
```

So, whereas an if statement *always* costs you, it's nearly free to set up a try/except block. But when an Exception actually occurs, the cost is much higher.
Moral:

- It's perfectly OK (and "pythonic") to use try/except for flow control,
- but it makes sense most when Exceptions are actually exceptional.

From the Python docs:

> **EAFP**
> Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many try and except statements. The technique contrasts with the LBYL style common to many other languages such as C.

2.7 regular expression

2.7.1 grammar

https://docs.python.org/2/library/re.html

'.'

(Dot.) In the default mode, this matches any character except a newline. If the DOTALL flag has been specified, this matches any character including a newline.

'*'

Causes the resulting RE to match 0 or more repetitions of the preceding RE, as many repetitions as are possible. ab* will match 'a', 'ab', or 'a' followed by any number of 'b's.

regexp = '.*\.txt' # the first . is the wildcard

2.7.2 re.match

```
re.match(searchterm, file)
```

re. match(*pattern*, *string*, *flags=0*)

If zero or more characters at the beginning of *string* match the regular expression *pattern*, return a corresponding match object. Return None if the string does not match the pattern; note that this is different from a zero-length match.

Note that even in MULTILINE mode, re.match() will only match at the beginning of the string and not at the beginning of each line.

If you want to locate a match anywhere in *string*, use search() instead (see also search() vs. match()).

2.8 list.sort

http://www.dotnetperls.com/sort-python
http://www.dotnetperls.com/sort-file-size-python
https://wiki.python.org/moin/HowTo/Sorting

2.9 import os module

```
try:
    files = os.listdir(path)   #try listing the files in the directory
```

```
try:
    full_filename = os.path.join(path, filename) # if we pass nonsense to the os., it will still crash,so try-except

if os.path.isfile(full_filename): # check whether the file exist

size = os.path.getsize(full_filename)    # get the size of each file
```