PIC 16, Winter 2018 – Preparation 2F

Assigned 1/17/2018. To be completed by lecture 1/19/2018.

Intended Learning Outcomes

By the end of this preparatory assignment, students should be able to:

- manipulate sequence elements compactly using built-in functions including `filter`, `map`, `reduce`, `enumerate`, `zip`, `reversed`, and `iteritems`, and perform the same tasks using more general features like data structure comprehensions and slicing, and
- compare variables as needed, choosing between identity and equality comparison operators correctly and chaining comparisons as appropriate.

Tasks

- ☐ Read 5.1.3
- ☐ Write a lambda function `square` that returns the square of a number. Then use the built-in `map` function to print out the squares of integers 1 through 10. (No for loops)
- ☐ Write a function `is_square(a)` that returns `True` if a is a perfect square (the square root of `a` is an integer) and `False` otherwise. There are several ways to test whether a number is an integer; try to figure it out using functions/operators you already know.
  Then use `filter` to determine which of the numbers in this list are perfect squares:
  `x = [484, 539, 566, 625, 686, 729, 784]`
- ☐ The functions `filter`, `map`, and `reduce` can make your code concise and possibly more computationally efficient. However, I find it difficult to memorize all these functions myself. The syntax of `for` loops is easy to remember, though, and it's general enough to handle anything these functions can. Write `for` loops to accomplish the same tasks as the `filter`, `map`, and `reduce` functions do in the following examples.

  ```
  def f(x): return x % 3 == 0 or x % 5 == 0
  print filter(f, range(2, 25))

  def cube(x): return x*x*x
  print map(cube, range(1, 11))

  def add(x,y): return x+y
  print reduce(add, range(1, 11))
  ```

  The downside in most applications is that `for` loops take several lines instead of one, but we can fix that using another general language feature. Use list comprehension to perform the job of `map` and `filter` (not `reduce`) in the previous task. Search for "list comprehension vs map" and you'll find list comprehensions to be considered better Python, and the speed is comparable.

- ☐ Read 5.6
- ☐ Again we have functions that are handy but can be accomplished otherwise.

  Do the same as:
  ```
  l = ['tic', 'tac', 'toe']
  for i, v in enumerate(l):
      print i, v
  ```

without the `enumerate` function. When you find yourself writing such code, consider using `enumerate` instead, as it is often more clear to other Python programmers.

```
questions = ['name', 'quest', 'favorite color']
answers = ['lancelot', 'the holy grail', 'blue']
for q, a in zip(questions, answers):
    print 'What is your {0}?  It is {1}.'.format(q, a)
```

Do the same as above without using `zip` – but in the future, if you find yourself writing such code, try to use `zip` instead. Note that you cannot produce the exact same output using only `print` and commas because commas produce spaces. We haven't covered the `format` method, but I'll bet you can figure out how it's working. Just try to follow the example.

```
l = range(1,10,2)
for i in reversed(l):
    print i,
```

(Note that the comma at the end of the `print` command suppresses the line break.) Do this without `reversed` by using slicing. This uses more memory, but can be useful when you don't need to iterate backwards over the entire list.

Now do the same as

```
knights = {'gallahad': 'the pure', 'robin': 'the brave'}
for k, v in knights.iteritems():
    print k, v
```

without using `iteritems`.

- Read 5.7.
- Predict and check the output of the following code:

```
knights = ["Arthur", "Lancelot", "Robin", "Bedevere", "Galahad",
"Gawain", "Ector", "Bors"]
print "Lancelot" in knights
print "Black Knight" in knights
x = [1, 2, 3]
y = [1, 2, 3]
z = x
print x is y
print x == y
print x is z
print x == z
print y is z
print y == z
x.append(4)
print x
print y
print z
```

Understanding that variables never store objects themselves - they only reference objects that "live" elsewhere in memory, independent of the variable - is very important, and is essential to understanding the difference between identity and equality. Please let me know if you are struggling with this, as this may be different in the programming language you are used to.

- Read 5.8
- Predict and check the output of the following, modified slightly from the examples that appear in the reading.

```
print (1, 2, 3)              < (1, 2, 4)
print [1, 2, 3]              < [1, 2, 4]
print (1, 2, 3, 4)           < (1, 2, 4)
print (1, 2)                 < (1, 2, 4)
print (1, 2, 3)             == (1.0, 2.0, 3.0)
print (1, 2, ('aa', 'b'))    < (1, 2, ('ab', 'a'), 4)
```

Although variables are dynamically typed in Python, the values still have types, and that's important for understanding the last bit of 5.8. Run these two lines.

```
print type(True)
print type('Pascal')
```

Now predict the outcome of these next few. Note the chaining discussed in 5.7!

```
print 'ABC' > 'C' < 'Pascal' < 'Python'
print True < 'Pascal'
print ('ABC' > 'C') < 'Pascal' < 'Python'
```

If you had trouble with the previous three, read 5.8 again.

It seems that 5.8 forgot to mention that booleans compare to integers just like in C++; not by the name of the type of variable. Or maybe they consider booleans a numeric type. Check this:

```
print False == 0
print False == 1
print True == 0
print True == 1
print True == 2
print True + 1
```

Now try these:

```
x = 1; print 1 < x < 3
x = 2; print 1 < x < 3
x = 3; print 1 < x < 3
x = 3; print (1 < x) < 3
x = 3; print (1 < x) < 0
```

- Read through "Timing Code Segment Execution" in Quick Guide to Python Performance. You'll practice this in your next assignment. The rest of the document is for future reference.
- If you have time, check out this supplementary video, which has some tips on using all these different profiling methods. But again, you'll only need `time.time()` or `time.clock()`, depending on your system, right now.