Lab 6: More Python

Dictionaries

- Dictionaries are created using curly-brackets
  - e.g. `A = { key1 : value1, key2 : value2 } ;` `A = {}`
- They consist of a set of Key-Value pairs
  - There are no restrictions of the values (they can be anything)
  - No duplicate keys are allowed, and a key must be **immutable**
    - Valid keys would be strings, tuples, numbers, etc.
- Dictionaries are like lists in that the are **mutable and unordered!**
  - `A[key1] = value3` → {key1 : value3, key2: value2}
  - `del A[key1]` → {key2: value2}
- To loop through a dictionary:
  - e.g `for key,value in _dict.items():`

Dictionary Methods

- Get list of keys: D.keys()
- Get list of value: D.values()
- Get list of (key,value) tuples: D.items()
- Remove all elements: D.clear()
- For more examples on lists, tuples, dictionaries:
  - https://docs.python.org/3/tutorial/datastructures.html

- Remember, like lists, if you modify a dictionary in a function it will be modified outside the function!

List Recaps

- Last week we learned that we can make lists using brackets
  - e.g `A = [1,2,3]` ; `A = [1, “two”, 3]` ; `A = []`
- You can loop through lists and modify existing lists

```
A = []
for x in range(0,6):
    if x % 2 == 0:
          A.append(x)
print(A) #→ [0, 2, 4]
```

- Remember that lists are soft-copies!
  - What I do: `A = B = [1,2,3]` ; `B.append(4)` ; `print(A)`

List Comprehension

- List comprehensions are a tool for transforming lists (or anything iterable, really) into another list
- An easier (and faster) way of doing the previous slides example

```
A = [ x for x in range(0,6) if x % 2 == 0]
```

- Kinda hard to read? Python supports line breaks between brackets and braces (so you can do this with dictionaries too)

```
A = [ x
      for x in range(0,6)
      if x % 2 == 0
      ]
```

Examples

- Change list values
  - `doubled = [n * 2 for n in range(0,10)  ]`
- Nested loops
  - `matrix = [[1,2],[3,4]]  ;  flattened = [n for row in matrix for n in row]`
- Dictionaries
  - `_dict = {1:2, 3:4} ;`
  - `flipped = {value: key for key, value in dict.items()}`
- Convert dictionary to list of values
  - _dict = {1:2, 3:4} ; dict_values = [val for val in _dict.values()]

Recall from last week: Function definition

- Any code that exists at the outermost indentation level is considered *global*
  - Better idea, use functions!
- Functions are defined using **`def`**

```
def main():                    ← ‘:’ indicates a change in scope

    print(“I ran from command line”)

if “__name__” == “__main__”:    ← Everything on leftmost line is global!!!

    main()
```

Nested Functions

```
def test():
    def test3():
        print("Hello!")

    test3()  #Runs successfully!

test() #Run successfully!
test3() #Fails - Why?
```

← Every time you use ‘:” you need to change your tab level ie. your scope!

Map

- Map will take the contents of a list and pass them to a function
  - This returns an iterable map object (<map object at 0x7fb769e976a0>) that you can use in, for example, a for-loop
  - This means that it isn't directly subscribable!
  - You can convert a map object to a list by using ***list(map_object)*** if you want to access it directly
    - Note: If you only plan to use it as an interable, don't convert it as it's faster not convert it
- Example: Convert string characters to int
  - one,two,three = map(int, ["1","2","3"]) → What happens if I make this **= B?**
    - If you know the number of values being returned, you can assign them to variables directly (this is true for functions as well)

Filter

- Filter is similar to map, except it extracts the element from the list that returns True from the passed function
  - Note: The same as map() it will return an iterable filter object unless you convert it to list using *list()*
- For example, get all values that divisible by 2

```
def f(x): return x % 2 == 0

F = filter(f, range(2,10))
print(F) → <filter object @ …>
print(list(F)) → [2, 4, 6, 8]
```

Lambda

- Lambdas are anonymous functions
  - Because they exist in the local scope of wherever you call them, you can pass them local variables
  - They are very handy to be used with *map()* or *filter()*
- Syntax
  - You can assign a lambda as a variable
    - square = lambda some_number: some_number * some_number
    - square(4) → 16
  - Or run it using map
    - X = list(map(lambda x: x*x, range(0,4)))
    - print(X) → [1, 4, 9]

Tricky stuff

- Some tricky things with lambda
  - `A = [ lambda x: x*x for x in range(0,3) ]`
    - `A[0]` → Is the lambda function at list index 0
    - `A[0](4)` -> Pass the lambda function a 4 (Output = 16)
    - What if you did A[1](4) ?
- You can pass in local variables to a lambda, but be careful!
  - `x = 10 ; A = lambda i: x*i` ← `This assumes ‘x’ exists!`
  - `print(A(1))` → What will this output?
  - `x = 20 ; print(A(1))` → What will this output?

Recall - File IO

- To read/write a file you use the following syntax:

```
with open(filename, params) as fp: ← fp is our file handle
    <Go here if file opens successfully>
<File auto-closes>
```

- The code under the *with* indentation is run, if and only if, the file is opened successfully. You should stick all file-specific code in this indentation level
- *params:* These are how you want to handle the file
  - ‘r’: read ‘rt’: read as text ‘rb’: read as binary ‘r+’: read and write
  - ‘w’: write ‘wt’: write as text ‘wb’: write as binary
  - ‘a’: append

Recall : File IO

- You can also loop through the lines of a file directly:

```
with open(filename, “rt”) as fp:
    for line in fp: #Does not remove ‘\n’
        line = line.rstrip() #Remove trailing ‘\n’
        output = handleFileRow(line)
```

- Read all file contents: **data = fp.read()**
- Read one line at a time: **data = fp.readline()**
- Read all lines into a list: **data = fp.readlines()**

Python 3 quirks

- Python 3 returns *iterators* or *generators* from it's list creation functions
  - If you were to print the result of a *map()* you would get an map object that is an iterator
  - So? That means when you assign variables it leaves the iterator!
    - **This is true to file IO as well**!

```
myint = map(int, ["1", "2", "3"])
print(list(myint)) → [1, 2, 3]
one, two, three = myint
print(list(myint)) → []
print(one,two,three) → 1 2 3
```

- Python 3 converts *int* to *long int* in the interpreter. Why does that matter?
  - If you use an integer in a for-loop, this slows down execution quite a bit
  - You can use the Python 3 built in array type, where you can assign the data-type
    - e.g. array('H', [1, 2, 3, 4, 5]) → This will create a array-list of unsigned short
    - https://docs.python.org/3/library/array.html

Practise Assignment

- You have been provided a comma-separated text file called **students.txt**
  - Each line contains: first_name, last_name, age, course1, … courseN
- Create a file called **parser.py** that does the following:
  - Calls a main() function when run from the command line
  - Read the filename from the command line and open it
  - Store the contents of each line in a dictionary in the following format:
    - {‘first’: first_name, ‘last’: last_name, ‘age’: age, ‘courses’: list_of_courses}
    - Save the dictionary in a list called **students**
  - Using map(), filter() and lambda create a list of tuples **student_courses** where:
    - (LAST_NAME, first_name, number_of_courses)
    - Only students who age is greater than 25
- Output the result to a file called **older_students.txt**