Introduction to Machine Learning

Lecture 2: More Python and Alice

Ian J. Watson

University of Seoul

University of Seoul Graduate Course 2019

KRF KOREA RESEARCH FELLOWSHIP
해외우수신진연구자유치사업

Housekeeping

- Today, we'll review python, but some housekeeping first
- Further issues with Thursdays
  - Oct. 26 will move to Mon., Oct 21 1-4 pm
  - If you can't make it, try to do the exercises at home, I won't mark you absent (but will mark you down if you don't complete the exercises!)
- Find the classroom link at `https://git.io/ml2019-2`
  - Complete all the tasks by the end of class and **remember to upload to github**
  - Ask if you need help
- Remember to log out!

Further Resources

- Some resources to help your programming skills (free to read online):
  - Think like a computer scientist
    `http://openbookproject.net/thinkcs/python/english3e/`
    - Gives a very good and slow overview of how to program
  - Automate the Boring Stuff with Python:
    `https://automatetheboringstuff.com/`
    - Gives very practical and usable examples of how programming can help you with everyday tasks
    - Particularly for researches, there are lots of places where a simple script can turn days of drudge work into a minute running a script
- For those who want/need homework, run through the exercises in these books and find a place in your current workflow that can be improved with programming

Lists reminder

- We can also have composite data types, that is, a type that holds other types
- A list holds several objects in a single data structure
  - So, the empty list (list with no objects) is `[]`
- Use square brackets with objects separated by lists

```
lst = [1, 2, 3, 4]
print("Length {}".format(len(lst))) # len Gives the length of l
print(lst[0])
```

- Individual objects can be accessed using square brackets and an *index*, starting at 0, and going to `len(list)-1`
- You can grow a list with append, it will grow in place

```
lst = [1, 2, 3, 4]
lst.append(5) # Now lst is [1, 2, 3, 4, 5]
lst[3] = 2 # Can reassign, now lst is [1, 2, 3, 3, 5]
lst[4] # would fail before the append statement
```

- You can + lists together to *concatenate* them

Lists and looping

- To loop over a list, we can use `for`
- `for a in l:`
  - `l` should be a list, then `a` is a new variable, which gets replaced with each list element in turn

```
for a in [1, 2, 3, 4]:
    print(str(a))
```

- prints "1", then "2", then "3", then "4"
- At the end, `a` will be filled with 4, but really, we shouldn't use list variables after the loop, this is a side-effect of python's *scoping rules* [some technical info. follows]:
  - *variable scope* refers to where a variable is allowed to be used, and where it gets destroyed. Python has *function scope*, that means variables are defined and exist for the extent of the function
  - In other languages, the loop variable would go *out of scope* at the end of the loop, this is part of *lexical scope*, where blocks define variable *lifetime*

Range

- We often want to get an index `i` to go from 0, 1, 2, 3, etc. giving n numbers out: `[0, 1, 2, 3, ..., n-1]`
- We can use the function `range` to do this:

```
l = []
for i in range(10):
    l.append(i)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- `range` can also start at a different number
  - `range(a, b)` will go from `a` to `b-1`

```
l = []
for i in range(3, 10):
    l.append(i)
```

```
[3, 4, 5, 6, 7, 8, 9]
```

Range, continued

- Finally, we don't have to increment by 1
  - In `range(a, b, c)` the list ends when the next item would be equal to b or more

```
l = []
for i in range(3, 10, 2):
    l.append(i)
```

```
[3, 5, 7, 9]
```

- We can use this when we want to skip through lists, taking every nth element, for instance

Ranges and list

- If theres some reason not use a simple for loop (for example we need to process two lists of the same length, or we want to change the list) we can use ranges to loop over lists

```
l = [1, 2, 3, 4]
for i in range(len(l)):  # i is an "index" to the list
    l[i] += 1 # add one to each element of the list
```

```
l1 = [1,2,3,4]
l2 = [3,4,5,6]
for i in range(len(l1)):
  # Now can you use elements of both lists
  print("{} {}".format(l1[i], l2[i]))
```

- In this case we could also use `zip(a, b)`, look up `help(zip)` if you are interested

enumerate

- The code on the last page is fine, but its more pythonic to use `enumerate` for this

```
l = [1,2,3,4]
for i, el in enumerate(l):
    l[i] = el + 1
```

- `enumerate` gives us two outputs for every element: the index (here `i`), and the element itself (here `el`)
- There are many ways to accomplish the same task!
- Pick a way that makes sense to you and use that
- Be aware of alternate approaches, if you find yourself writing too much code, there's probably a simpler way to do it!

Functions

- We've been using lots of functions, like `len`, or `format`
- You pass the function *arguments* to be used inside the function
  - `len(lst)`, `math.log(10, 2)`
- You define with `def`, you need to name it, and give a *parameter list*
  - For each *parameter* of you function, you expect the user of your function to pass an *argument*
- You can do work inside the function, then you can `return` a value back to the user

```
def add_3_numbers(a, b, c):
    return a+b+C

d = add_3_numbers(1, 2, 3) # d will be set to 6
```

Functions (continued)

- Lets look a closer at the function `def`
- `def f():`
  - This defines a function of no parameters. You don't pass any arguments

```
def f():
 return 1

f() # 1
```

- `def f(a):` this defines a function of 1 parameters
  - When you pass an argument, whatever was passed gets *bound* to a inside the function

```
def f(a):
  return a+1 # I can use a inside the function
```

Functions, return

- `return` **immediately** *exits* the function, and returns the value to the *caller*, the user of the function
- You can `return` anywhere inside the function
  - If you have special conditions, good to return early, then process the data knowing that that condition can't hold
- The factorial function is defined $(n! = 1 \times 2 \times 3 \ldots \times n)$
  - Or: $0! = 1$, $n! = n \times (n-1)!$, this is a *recursive* definition, it defines the function in terms of itself, plus a stopping case, check the stopping case first, then process the function:

```
def factorial(n):
    if n == 0: return 1
    return n * factorial(n-1)

factorial(5) # 120
```

Functions (continued)

- Function calls pass a *reference* to the parameter, that is, it takes a *shallow* copy of the argument
  - That means, if you change integers in the function, outside the function, at the *call site* they won't change, but *mutable* objects, like lists or dictionaries *will* change

```
a = 4
def add_4_toi(i):
  return 4+i

add_4_toi(a) # We're ignoring the return value here
a # a is still 4
```

```
a = [1,2,3]
def add_4_tol(l):
  l.append(4)

add_4_tol(a)  # inside the function, l /binds/ to the list a
a # Now a is [1, 2, 3, 4]
```

Exercises from Last Week

Lets go through the exercises from last week

This week : Alice in Wonderland

- There's a text file `alice.txt` in the repo this week
- It contains the complete text of Alice in Wonderland
- As a python exercise and first effort to analyse some data, we'll write some functions to process the text and develop some simple analyses
- E.g. we'll look at the number of different words used, the most and least popular words in the text, and the average number of times a separate word is used

Exercises in this weeks notebook