MATH 580A 6/30 Notes

1 Recap and Introduction

Last class we covered strings and dictionaries, which were the last two major common data structures in Python. We saw how to format strings, use dictionaries to store more complicated data, and had a brief exposure to representing directed graphs in Python.

Today we will cover some miscellaneous useful topics in Python and conclude with a large example on LZW compression. This will complete the list of basic topics and syntax in Python. Starting next week we will cover more algorithmic topics, starting with graphs. These topics will provide us with lots of examples to reinforce programming in Python.

2 Lambda Functions and Functions as Arguments

We've worked with functions quite a bit at this point. We've seen how to define a function and also seen different argument definitions, including positional arguments, keyword arguments, and taking in an arbitrary positional argument list. We will now look at one more way to define functions called *lambda functions*. These are short one-line expressions that define a function without a name. They are usually used as a convenient way to write small functions without having to give them a name.

Listing 1 demonstrates a multitude of ways to define and use lambda functions. Line 1 shows the basic syntax for defining a lambda function. We use the `lambda` keyword, followed by a list of argument names, followed by a colon `:`, and then the function body. Unlike a regular function, there is no `return` keyword and whatever value we write in the function body gets returned. Also, unlike a regular function, we can't execute multiple computations or use for loops or if statements. Notice that in line 1, we actually define the function and assign it to a variable called `double`. The reason we can do this is that in principle, functions are just a type of value, like strings, numbers, lists, and dictionaries. Thus, we can assign it to a variable. Once assigned to `double`, we can use the parentheses notation `double(10)` to call the function, passing in the argument 10.

Lines 4 through 8 are an example of a lambda function that takes two arguments `v,w` and computes their cross product. Here we assume the arguments are both lists of length 3. Although we said that lambda functions are typically one line, the actual requirement is that the function body consists of a value that's returned. In this case, since we can define lists across multiple lines, we have a multiline return value which is a single list.

Lines 11 and 12 show another example of how functions are a type of value just like strings and numbers. We define a lambda function that takes two arguments, a function `f` and a value `x` and returns `f` applied twice to `x`. Observe that the function `double`, acts as any other value, and can be used as the argument in `apply_twice`.

Line 14 shows a more complicated lambda function that combines its two input functions, composing one after another. In order to define this, the return value of `compose` is itself a

```
 1  double = lambda x: x * 2
 2  print("double(10):", double(10))
 3
 4  cross_product = lambda v, w: [
 5    v[1]*w[2] - v[2]*w[1],
 6    -v[0]*w[2] + v[2]*w[0],
 7    v[0]*w[1] - v[1]*w[0],
 8  ]
 9  print("cross_product([1,1,-1], [0,2,1]):", cross_product([1,1,-1], [0,2,1]))
10
11  apply_twice = lambda f, x: f(f(x))
12  print("apply_twice(double, 10):", apply_twice(double, 10))
13
14  compose = lambda f, g: (lambda *args: g(f(*args)))
15  print("compose(double, double)(10):", compose(double, double)(10))
16  print("compose(cross_product, sorted)([1,1,-1], [0,2,1]):",
17        compose(cross_product, sorted)([1,1,-1], [0,2,1]))
18
19  list = [[1,1], [-1,0], [1,0], [0,-1]]
20  list.sort(key = lambda coord: math.atan2(coord[1], coord[0]))
21  print("list:", list)
22
23  functions = [double, lambda x: x+1, abs]
24  y = -7
25  for fn in functions:
26    y = fn(y)
27  print("y:", y)
```

Listing 1: Function Examples.

function. It is a function that takes an arbitrary number of positional arguments and forwards those along into `f`.

Lines 19 to 21 gives a common practical application of lambda functions. Often you will have lists of some type of interesting object and wish to sort the list using the built in `sort()` function. However, Python doesn't know how to compare elements in your list and you'll need to provide a function as the `key` argument. The function should take an element of the list and return some value that Python does know how to compare, usually a number or string. In this case, we return the angle from the x-axis for each of the coordinates and sort the coordinates by their angles. Note that the `key` argument in Python's `sort()` function is a keyword-only argument so you must type out the `key=` section.

Finally, lines 23 through 27 show one more example of the versatility of functions. You can put them in lists (and also dictionaries!) and iterate over and call them.

Do exercise 1. The first exercise asks you to compute the fixed points of a function passed in. The second exercise asks you to compute a list of maximal elements among the passed in list, using a custom comparator function. The final exercise asks you to take a list of lists and flatten it.

3 List Comprehensions

There is a syntax for constructing lists that is very compact and convenient. In mathematics, one might write $\{x^2 : x \in \mathbb{Z}\}$ to denote the set of all square integers. Python has a syntax similar to this called a *list comprehension*. The syntax for this looks like `[<variable> for <variable> in <iterable>]`, where `<iterable>` is any sort of object you can loop over with a regular for loop.

```
1  sample = [1,2,4,8,16,32]
2  doubled = [2*i for i in sample]
3  print("doubled:", doubled)
4
5  multiples_of_four = [i for i in sample if i%4 == 0]
6  print("multiples_of_four:", multiples_of_four)
7
8  def add(list1, list2):
9    return [list1[i] + list2[i] for i in range(min(len(list1), len(list2)))]
10
11 def add_(list1, list2):
12   return [x+y for x,y in zip(list1, list2)]
13
14 print("add([1,2,3], [-1,-2,-3])", add([1,2,3], [-1,-2,-3]))
15 print("add_([1,2,3], [-1,-2,-3])", add_([1,2,3], [-1,-2,-3]))
16 print("zip([1,2,3], [-1,-2,-3])", zip([1,2,3], [-1,-2,-3]))
17 print("list(zip([1,2,3], [-1,-2,-3]))", list(zip([1,2,3], [-1,-2,-3])))
```

Listing 2: List Comprehension Examples.

Listing 2 shows some examples of using list comprehensions. Line 2 is one of the simplest ways of writing a list comprehension. It defines a new list whose elements are those in `sample` except that each element is doubled.

Line 5 demonstrates how list comprehensions can also be used to filter for elements that satisfy a certain condition. Here, we use the condition `if i%4 == 0` in order to obtain a sublist of `sample` that consists only of the values divisible by 4.

Lines 8 and 12 show two ways to write a function that takes in lists `<list1>` and `<list2>` as arguments and returns a list of the component-wise sum of the two. The second definition demonstrates the built-in Python function `zip()` which takes two lists and returns a list where elements are paired up together, truncating to the length of the shorter list. For example, `zip([1,2], ["a","b"]) = [(1,"a"), (2,"b")]`.

List comprehensions, like lambda functions, are primarily included in Python as a convenient way to implement certain functionality in fewer lines of code. You could equivalently start with an empty list and use a regular for-loop to append elements, but would often have to write a fair bit more code to do so.

4 Miscellaneous Data Structures

Although we have covered the main data structures in Python, there are a couple more built into the language. There is the *tuple* which is similar to a list, but immutable. That is, once a tuple is created, you cannot modify the contents of it or add or remove to it. There is the *set* data structure which is a container that holds values, but is unordered and deduplicates values. Finally, there is the *frozenset* data structure which is analogous to a tuple in that it is a set that is immutable.

Listing 3 illustrates the use of these data structures and also demonstrates a couple pitfalls. Line 1 shows the basic definition of a tuple. It differs from lists in that you use parentheses instead of square brackets to define the tuple. Lines 5 and 6 show invalid operations on a tuple as it is immutable. You can uncomment these lines to see what errors Python will raise.

Lines 8 through 15 show why immutability is useful and why you might use a tuple over a list. Dictionary keys need to be *stable*. You can't use a list as a key because the contents of a list might change, rendering it invalid. To get around that, you can instead use tuples as keys because their contents will not change. Lines 14 and 15 show invalid operations. Because lists and dictionaries are mutable, they are not valid to use as keys.

The basic definition for sets is demonstrated on lines 17 through 19. If you call `set()` with no arguments, you get back an empty set. To define a set with some initial elements, you can use curly braces, similar to in mathematics. This differs from the definition for a dictionary because there are colons creating key-value pairs. Another way to define a set with elements is to pass in a list-like object as an argument to `set()`. Python will convert it into a set, removing any duplicate entries in the process. To add or remove elements, use the `add()` or `remove()` functions as on lines 24 through 28. Notice that if you add an element that already exists in a list, nothing will happen.

Lines 31 through 35 demonstrate some mathematical set operations you can do. First, the `<` operator is used to check for strict set inclusion. You can use `<=` to check for set inclusion with equality. The `|` operator computes the union of two sets and the `&` operator computes the intersection of two sets.

Finally, lines 37 through 42 demonstrate frozensets. Because a frozenset is immutable, you will generally call `frozenset()` with a list-like object as the argument to create the immutable object. Frozensets can be used as dictionary keys because they are immutable but regular sets cannot, as seen on line 42.

Do exercise 2. Exercise 2 is a fairly substantial exercise about solving a word game called Spelling Bee!

```python
coordinate = (0,1)
print("coordinate:", coordinate)

# Invalid code:
#coordinate[0] = 1
#coordinate.append(0)

points = {}
points[coordinate] = "A"
points[(1,-1)] = "B"
print("points:", points)

# Invalid code:
#points[[-1,0]] = "C"
#points[{}] = "D"

emptyset = set()
integers = {1,2,3,1,2,3}
deduped_letters = set("SYSTEMATIC")
print("emptyset", emptyset)
print("integers", integers)
print("deduped_letters", deduped_letters)

integers.add(4)
integers.add(3)
print("integers after adding:", integers)
integers.remove(2)
print("integers after removing:", integers)
more_integers = set(range(100))

print("integers < more_integers: ", integers < more_integers)
print("integers < integers: ", integers < integers)
print("integers <= integers: ", integers <= integers)
print("union:", integers | deduped_letters)
print("intersection:", deduped_letters & set("ORANGE"))

d = {}
d[frozenset(integers)] = True
print("d:", d)

# Invalid code:
# d[integers] = False
```

Listing 3: List Comprehension Examples.