MATH 580A 6/23 Notes

1 Lists in Detail

Although our variables have so far mostly taken on numerical values, we saw a couple examples last time of other values you can store in a variable: *lists* and *dictionaries*. These are a couple built-in *data structures* in Python and let you store more complicated types of data. Let's first dive into lists (also called *vectors* in C++).

A list is a sequential list of values and can be created using the square bracket notation. For example, `powers = [1,2,4,8]` creates a list called `powers` with 4 elements. You can *index* a list in order to retrieve specific values in it. This also uses the square bracket notation. As mentioned last time, Python arrays are *0-indexed* so for example, `powers[2]` would return the value `4` which is the *third* element of the list.

Unlike other programming languages, Python provides some pretty powerful language features for extracting values out of a list. First, you can also use negative indices in order to index elements from the back of the list. For example, `powers[-1]` would return `8` and `powers[-4]` would return `1`. However, the indices do not wrap around, so indices too large such as `powers[10]` or too small such as `powers[-10]` will result in errors.

In addition to negative indices, Python also supports *slices* which let you pick out a range of the list and create a new sublist out of it. The notation for this is to use a colon and specify the beginning and ending indices of the slice. The beginning is inclusive and ending is exclusive. For example `powers[0:4]` would return the entire `powers` list and `powers[1:3]` would return the sublist `[2,4]`. Finally, you may omit the beginning and/or ending indices and Python will use the beginning or ending of the list respectively. Thus, `powers[:3]` would return the sublist `[1,2,4]` and `powers[:]` would return a copy of the entire list.

Python also lets you add and remove values to a list. You can use the `list.insert(<index>, <value>)` to insert a value at a specific index in a list and `list.pop(<index>)` to remove a value at a specific index in the list. Similar to indexing, you can pass in negative values for `<index>` and those will count from the back of the list. One more fundamental operation on lists is to return the *length* of the list which is the number of elements it contains. You can use `len(<list>)` in order to return the length. For example, `len(powers)` will return 4.

Building on top of these basic operations allows us to create more complicated functions. In fact, Python has many built-in functions to help with lists such as `list.reverse()` and `sum(<list>)`. Listing 1 shows some custom implementations of the built-in functions Python provides, demonstrating how to build on top of the more basic operations.

A common pattern in some function computations is to do some operation for each element in a list and then return the final value. I often will call this variable `ret`, short for "return value" but this is non-standard. In some contexts, you may see such a variable referred to as an *accumulator*.

Another note is that our implementation of the `reverse()` function operates on the list *in-place*. What this means is that instead of creating a new list, it modifies the input list directly.

```python
# Append <value> to the end of <list>.
def append(list, value):
  list.insert(len(list), value)

# Remove the first occurence of <value> from <list>.
def remove(list, value):
  for i in range(len(list)):
    if list[i] == value:
      list.pop(i)
      return

# Count how many times <value> occurs in <list>.
def count(list, value):
  # short for "return value"
  ret = 0
  for i in range(len(list)):
    if list[i] == value:
      ret += 1
  return ret

# Reverses <list> in-place.
def reverse(list):
  for i in range(len(list)):
    value = list.pop(-1)
    list.insert(i, value)

# Returns the sum of all the elements in <list>.
def sum(list):
  ret = 0
  for i in range(len(list)):
    ret += list[i]
  return ret
```

Listing 1: Reimplementing List Functions.

This touches upon a concept known as *mutability* in Python. As mentioned earlier, a list can be thought of as a container and the contents of the list can be modified. Container-like objects in Python with contents that modified are known as *mutable*. Numbers on the other hand don't have any inner contents to modify and are *immutable*. In practice, what is most important to understand regarding this is that mutable objects have be referred to by multiple variables at the same time.

Listing 2 demonstrates a potential pitfall involving mutable objects. Line 2 assigns the list in `list1` to `list2`. Before, we talked about variables kind of like locations or addresses in a warehouse. In this case, there is a single actual list, and both `list1` and `list2` are addresses that point to the same underlying list. So when we modify the list using `list2.append(100)`, the list pointed to by `list1` is also modified because it is in fact the same list. Sometimes this is

desirable and sometimes it isn't. If you want a separate copy of a list, you will want to use the `list.copy()` method. Alternatively, the slice notation `list[:]` will also produce a copy.

```
1 list1 = [1,2,3]
2 list2 = list1
3 list2.append(100)
4 print(list1) # outputs "[1,2,3,100]"
```

Listing 2: Demonstrating List Mutability.

Lists can also be *nested* inside one another. For example `grid = [[1,2,3], [4,5,6], [7,8,9]]` would create a list of lists and each of the inner lists contains 3 numbers. You could then index this list using two square bracket indices. For example, `grid[1][2]` would return the number 6. The first index `1` would pick out the second row, and then the second index `2` would pick out the third element in that row, which is 6. Such a list of lists is also often called a *2-dimensional list* and is a common way of representing a 2-dimensional grid of data. You can also have a list of lists of lists as well to represent 3-dimensional data and so on.

Listing 3 is demonstrates how to use multidimensional lists to represent image data and how to modify that data and produce a box blur effect. We use an external library called `matplotlib` that allows us to load images as a multidimensional list of numbers between 0 and 1. An image is a 2-dimensional list of *pixels* where a pixel is a list of 3 numbers between 0 and 1, representing the values of the red (R), green (G), and blue (B) channels respectively. A fully red pixel has RGB values `[1,0,0]`, a fully white pixel has RGB values `[1,1,1]`, a gray pixel has RGB values `[0.5,0.5,0.5]` and so on. Thus, the image we load is represented as a 3-dimensional list, where the dimensions are the row, column, and channel in that order.

Box blurring is the simplest blurring algorithm to implement. For a specified radius r, we replace each pixel `image[row][column]` with the average of the pixel values of all neighbors with L^1 distance less than or equal to r. That is, the average of all pixels in a box of side length $2r + 1$ centered at `(row, column)`, which is where the name of the algorithm comes from. A little bit of care needs to be taken near the edges of the image and these *edge cases* are accounted for on line 23. Observe that Python allows *chained comparisons* where you can write `a < b < c` and this is equivalent to checking `(a < b) and (b < c)`. This is not a common feature in programming languages and Python is special in this regard.

Another small detail is that on lines 19 and 20, we are using a version of `range()` that does not start at 0 but instead starts at `-radius`. The `range()` function can actually be called with 1 to 3 arguments. When it is called with just 1 argument, `range(<end>)` will let you loop from 0 up to `<end> - 1`. When it is called with 2 arguments, `range(<begin>, <end>)` will loop from `<begin>` up to `<end>-1`. Finally, when it is called with 3 arguments, `range(<begin>, <end>, <step>)` will loop as `<begin>`, `<begin> + step`, `<begin> + 2*step`, stopping right before it exceeds `<end>`.

This program is also a good example for analyzing *algorithmic complexity*. Often, you will see algorithms have their worst or average case performance written in big O notation such as $\mathcal{O}(x^2)$ or $\mathcal{O}(x^3)$. In general x is a measure of the size of the input and saying that the algorithm has worst case $\mathcal{O}(x^2)$ indicates that the runtime of the program (measured in the number of operations executed) is bounded above by Mx^2 for some fixed constant M when x is sufficiently large. The big O notation gives us a way to qualify how efficient certain algorithms are.

Let us take a look at the `box_blur()` function. How can the input size vary for this function?

```python
import matplotlib.pyplot as plt

def average(list):
  return sum(list)/len(list)

# image is a 3-dimensional array and indexed as [row][column][channel]
def box_blur(image, radius=1):
  height = len(image)
  width = len(image[0])
  channels = len(image[0][0])

  ret = []
  for row in range(height):
    new_row = []
    for column in range(width):
      new_pixel = []
      for channel in range(channels):
        neighbors = []
        for d_row in range(-radius, radius+1):
          for d_column in range(-radius, radius+1):
            row_ = row + d_row
            column_ = column + d_column
            if 0 <= row_ < height and 0 <= column_ < width:
              neighbors.append(image[row_][column_][channel])
        new_pixel.append(average(neighbors))
      new_row.append(new_pixel)
    ret.append(new_row)

  return ret

image = plt.imread("sample.png").tolist()
plt.imshow(box_blur(image))
plt.show()
```

Listing 3: Simple Box Blur Implementation

The `image` passed in will be a 3-dimensional list of size `height` $\times$ `width` $\times 3$ and so our input size is in terms of the height h and width w. In addition, there is another parameter which is the blur radius r. Now, how does the number of operations executed in the function vary based on h, w, and r?. Although the function body is only about 20 lines long, there are multiple nested for-loops. As a general rule, the total number of operations is approximately the product of the size of all the for-loops when nested together. In this case, the number of operations is $\mathcal{O}(3hw(2r+1)^2)$. In the big O notation however, we are not concerned with constants or lower order terms, and we can thus write the efficiency of the box blur algorithm as $\mathcal{O}(hwr^2)$. What this tells us is that if we double the height or width, we should expect the program to take about twice as long to run, but if we double the radius, it'll take about four times as long to run. In

general, we won't be able to avoid the h and w factor in a blur function if we want to examine each pixel when blurring. Thus, if we were to try to design a more efficient algorithm, we might look at seeing if we can make the algorithm depend linearly on r instead of quadratically. This is in fact possible, because the box blur is what is known as a *separable filter* and you will implement this version of it in the first assignment.

Do `exercise1.py` which involves writing selection sort. Analyze the algorithmic complexity.

2 A Little More on Flow Control

We've seen the if-else statements and the for-loop now. We've also looked at a couple variants on looping that we can do when using the `range()` function in the for-loop. There are some more variants and tools that let us control how our program chooses what operations to execute and we'll look at an example of these now.

Listing 4 simulates a variant of the coupon collector problem. In this problem, there are `NUM_COUPONS` different types of coupons to collect, labeled from `0` to `NUM_COUPONS-1`. The collector has a specific set of `NUM_DESIRED_COUPONS` that they want to collect. They continue to roll and collect a random coupon number until they've collected all the coupons they are looking for. At the end, we print out the distinct coupons they collect in order and how many tries it took for them to collect the coupons they wanted.

Lines 10 and 25 shows a different flow control tool called the *while-loop*. This is similar to for-loop in that it runs the code in the loop body multiple times. Unlike the for-loop, there is no loop variable that changes in each iteration. Instead there is a loop *condition*. As long as the loop condition is true, the loop body will keep running. In line 10, the condition is `True` which means the loop will run infinitely! That is, unless we explicitly execute the `break` statement on line 17 to exit the loop. The `break` statement can be used in for-loops as well to exit early and if there are multiple nested loops, it only breaks out of the innermost loop. Be careful of `while True:` loops because it can easily cause the program to be stuck in an infinite loop. Here we used it to illustrate the `break` statement. Line 25 shows a more typical case of using a while-loop. The loop body will continue to run as long as the number of desired coupons collected is less than the total number of desired coupons.

Lines 13 and 30 also illustrate another statement related to loops (both while-loops and for-loops). The `continue` keyword is similar to the `break` keyword, except instead of forcing the loop to exit, it forces the loop to move on to the next iteration, even if there would have been additional statements or operations to execute otherwise.

Two more Python language features are illustrated in line 31. If you want to chain multiple `if-else` statements together, you can use the `elif` keyword which is short for "else if". The general pattern is an `if` statement followed by some number of `elif` statements followed by an `else` at the end. Python will start from the top and check the conditions one by one until it finds one that is satisfied and execute the code in that block. The other language feature illustrated is the `in` keyword. When writing `<value> in <list>`, this is a condition that asks for whether or not the list contains a specific value. Although it is a single line of code, the execution time does depend on the length of the list since Python needs to go through the list one element at a time to check for the value.

Finally, line 39 illustrates a different variant on for-loops. Notice that there is no `range()` function here. Instead we just supplied the variable `collected` which is a list of numbers. This

```python
import random

NUM_COUPONS = 10
NUM_DESIRED_COUPONS = 5

desired = []
collected = []

# First determine which coupons are desired
while True:
  coupon = random.randint(0, NUM_COUPONS)
  if coupon in desired:
    continue
  else:
    desired.append(coupon)
    if len(desired) == NUM_DESIRED_COUPONS:
      break

desired.sort()
print("Desired coupons: ", desired)

# Collect coupons.
tries = 0
num_desired_collected = 0
while num_desired_collected < NUM_DESIRED_COUPONS:
  coupon = random.randint(0, NUM_COUPONS)
  tries += 1

  if coupon in collected:
    continue
  elif coupon in desired:
    collected.append(coupon)
    num_desired_collected += 1
  else:
    collected.append(coupon)

print("Tries taken:", tries)
print("Collected coupons in order:")
for coupon in collected:
  print(coupon)
```

Listing 4: Coupon Collector Problem Simulation

type of for-loop will iterate over all the elements in the list. On the ith iteration of the loop, the value of the loop variable `coupon` will be `collected[i]`. For-loops in Python are actually quite a bit more general than for loops in other popular programming languages like C++ and Java.

They can be used to loop over a general type of object in Python called an *iterator*. These are more advanced and we may omit discussing them. It often suffices to understand the common usages of the for-loop that we have been writing.

3 Advanced Ways of Defining Functions

Last time, we learned how to define functions. We'll now cover a few more advanced ways that functions can be defined. In practice, you likely won't need to write such functions often, but it is useful to know as it will help with understanding how to call functions provided in the Python standard library or provided by other packages such as NumPy or Matplotlib.

In Python, there are *positional* and *keyword* arguments passed into functions as input. The order of positional arguments matters, whereas keyword arguments have to be specified using the `<variable_name> = <value>` notation. Functions can also optionally take on arbitrary additional positional and keyword arguments through `*args` (short for "arguments") and `**kwargs` (short for "keyword arguments").

```
import random

def uniform(lower = 0.0, upper = 1.0):
  return random.random() * (upper - lower) + lower

print(uniform())
print(uniform(0.9))
print(uniform(10, 100))
print(uniform(upper=20))

def cartesian_product(*lists):
  if len(lists) == 0:
    return [[]]
  else:
    ret = []
    for list in cartesian_product(*lists[1:]):
      for element in lists[0]:
        ret.append([element] + list)
    return ret

print(cartesian_product([1,2,3]))
print(cartesian_product([1,2,3], [1,2,3], ["a", "b", "c"]))

def print_kwargs(**kwargs):
  for argument_name in kwargs:
    print(argument_name, "=", kwargs[argument_name])
print_kwargs(text = "Hello", pi = 3.14159, e = 2.7182818)
```

Listing 5: Some Function Definitions

Listing 5 demonstrates some of these function definition features. The exact manner in which these functions are implemented is not so important. However, notice that `uniform()` is defined with two arguments that have default values. These arguments are simultaneously positional and keyword arguments. Lines 6-8 show how to call the function supplying the input based on position. Since there are default values for `lower` and `upper`, one or both of them can be omitted. Line 9 shows how to call the function, treating `upper` as a keyword parameter.

Next, `cartesian_product()` is an example of a function that takes an arbitrary number of positional arguments based on the `*lists` argument. Lines 21 and 22 demonstrate calling the function with 1 list and 3 lists respectively. The built-in `print()` function also can take an arbitrary number of positional arguments.

Finally, `print_kwargs()` is an example of a function that can take an arbitrary number of keyword arguments. In practice, this type of function isn't used very often, but it is included here for completeness.

Complete `exercise2.py` and `exercise3.py`. The former involves using the Euclidean algorithm to compute the greatest common divisor of a set of numbers, and the latter involves implementing a simple image correction technique known as gamma correction.