MATH 580A 6/28 Notes

1 Recap and Introduction

Last class we covered the *list* data structure in detail, seeing various operations on lists and how to use multi-dimensional lists to process data. We gave a real-world computing application of multi-dimensional lists in the form of box blurring images. After covering lists, we took a look at a some more flow control tools in the form of while-loops, `break` and `continue` statements. Finally, we learned about different ways of defining functions. We saw the difference between positional and keyword arguments and saw how to define a function that can accept an arbitrary number of arguments.

Today, we will cover two main topics: strings and dictionaries. Strings are the basic objects used to store and handle text in Python and to a first approximation, act as lists of letters. Dictionaries are a more complicated data structure that allows you to store arbitrary data based on a *key*. Unlike lists which are a linear order of values and are indexed by the numbers 0, 1, 2, and so forth, dictionaries can be indexed by nearly any type of value. For example, you could write `dict["hello"]` or `dict[(1,2)]` or `dict[-100]` and so on. At the end of today's class, we will also discuss some details about how numbers and dictionaries actually work "under the hood". What we typically work with in Python are abstractions the language has provided us. Some knowledge about how it is implemented is helpful for when the abstractions break down in edge cases and we need to understand why.

We are also building up towards one major example application for next class which is a compression algorithm for text known as LZW. You may have heard of some compression schemes such as Huffman encoding, which can achieve optimality if you know the probability distribution on letter sequences. This compression algorithm is similar in that it doesn't lose any information during the compression, but differs in that it builds up a dictionary of commonly occurring strings along the way as opposed to doing any sort of up front analysis.

2 Strings

Strings are an object representing text in Python. Some languages have a separate *character* object for single letter strings, but in Python there is no distinction. The concept of a character is still useful and you can think of strings as lists of characters. However, strings have some additional methods specific to text and some list methods such as `sum()` don't make sense on strings.

In Python, there are some subtleties in defining strings. Listing 1 demonstrates some of them. Line 1 is the standard way of defining a string in Python. It consists of some text enclosed in double quotation marks. Line 4 is a variant in Python provided for convenience. If you prefer, you can use single quotation marks to define strings rather than double quotation marks.

Line 7 demonstrates *escape characters* in Python. A *newline* for example isn't typeable (splitting a string across multiple lines directly doesn't work). In order to have your text be split across multiple lines, you will need to use the newline escape character `\n` in your string. This is similar to control sequences in LaTeX except there are a standard predefined number of them in Python. For example, if you are using double quotation marks to delimit your string, then you will need the escape character `\"` to display a double quotation mark in your string. Finally, two backslashes in a row `\\` will display the backslash character itself.

Lines 10 through 13 demonstrate what to do if you have a really long string that you want to split across multiple lines in the code for readability reasons. As mentioned earlier, you cannot directly define a string across multiple lines. However, there is a special escape character that allows you to split a string definition over multiple lines. By writing a backslash character at the end of the line, it continues the current line of code on the next line. In fact, this works not just for strings but for long lines of code in general. You can split it over two or more lines using a backslash at the end of the line.

Finally, lines 16 to 19 show a special syntax for defining multi-line strings. You need to start and end the string with 3 double quotation marks. When using this syntax, all newlines are preserved so in this example, we end up defining a string with 4 lines total.

```
1  s = "It was the best of times. It was the worst of times."
2  print(s)
3  
4  s = 'You can use single quotes as well.'
5  print(s)
6  
7  s = "Backslashes are escape characters.\n\"\\n\" and '\\\"' are some examples."
8  print(s)
9  
10 s = "Really really really really really really really really\
11 really really really really really really really really really\
12 really really really really long strings can be cut across\
13 multiple lines using backslashes too."
14 print(s)
15 
16 s = """Triple quotes are used as multiline strings.
17 Newlines and
18     formatting are
19   preserved."""
20 print(s)
```

Listing 1: String Definitions.

Next, let us take a look at some string operations. They are similar to lists in many ways. Listing 2 highlights some of these common operations. Lines 1 and 4 demonstrate how to concatenate strings to form longer ones. In line 8 we see that the same list indexing syntax works for strings as well. You can use the `len()` function to obtain the length of a string. These are some of the list operations that also operate on strings.

Lines 13 and 14 show a couple useful functions for converting English letters in a string to all uppercase or all lowercase. These are often helpful if you're reading some input text and want to standardize to avoid having to distinguish between lowercase and uppercase letters.

Lines 16 to 20 demonstrate the `split()` function and its "inverse" function `join()`. The `split(<separator>)` function takes an argument which is used to split the string into a list of strings. In our example, we split `"06/28/2022"` according to the separator `"/"` which will give us a list of 3 strings `["06", "28", "2022"]`. The separator can be a string of any length, including the empty string `""`. The inverse function `<string>.join(<list>)` takes a list of strings and joins them together, using the calling string as the separator. In our example here, we join together the coordinates using the separator `", "` to obtain the string `"-1, 3, -8, 12"`.

Finally, lines 22 to 31 demonstrate substring comparisons. The `in` keyword can be used to check if one string is contained inside another. For example, on line 28, `"OWE" in "TOWER"` evaluates to `True` because `"TOWER"` contains `"OWE"` as a contiguous substring. Lines 24 to 26 demonstrate a small program used to print out all the letters that appear in a given word. Lines 30 and 31 demonstrate a similar function `<string>.find(<substring>)`. The difference between using `.find()` and the `in` keyword is that `.find()` will return the beginning index of the first occurrence of the substring if it exists. If the substring doesn't exist, then it will return -1.

One last aspect of strings that we will cover is formatting. Often, you will compute numerical values but printing them directly is not desirable and you'd like to round to a certain number of significant figures, or include commas to separate the thousands figures, etc. Listing 3 highlights how to do some of these formatting operations with strings.

We've briefly talked about f-strings previously but line 4 demonstrates how to use them once more. An f-string lets you mix literal text and the values of variable into a single string. You need to prefix the string with `f"` instead of the usual `"` to begin an f-string. Inside an f-string, { and } are treated specially and a variable inside will have its value printed out inside the string. If you need to print the actual curly brace characters, you can double them up. That is, `{{` and `}}` will print the curly brace characters themselves. Observe that in our example, any type of variable can be printed out in f-strings, not just numbers but also lists and other strings.

Lines 9 through 11 demonstrate several ways of formatting numbers. Inside an f-string, to format a number a specific way, add a colon `:` character after the variable name and then a format specification after the colon. The full range of specifications you can write is quite large and is detailed in the official Python documentation. These examples should give you a sense of what's possible with format specification, but is by no means exhaustive. A quick note is that writing an integer in binary as on line 11 is to write it using 0's and 1's. We will discuss binary representations in some more detail later.

Finally, lines 13 through 20 show how to convert from strings to numbers and back. The `float()` function takes a string and converts it to a *floating-point number*, that is, a not necessarily integral number. The `int(<string>, <base>)` function takes a string and converts it to a number. It can optionally take a base argument which specifies what base to interpret the string in. Finally, lines 19 and 20 show how to convert a number into a string. Line 20 in particular converts an integral number to its binary representation. The difference between this function and the f-string approach on line 11 is that `bin()` will prepend `0b` to the string it returns.

Do exercises 1, 2 and 3. Exercise 1 has two different mini-exercises involving string operations. Exercise 2 involves generating all the permutations of a list as a review of lists and recursion. Finally, exercise 3 asks you to format some numbers and requires learning to read the official Python documentation.

```python
s = "Before" + " " + "and" + " "
print(s)

s += "after"
print(s)

s = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
print(s[6:10])
print(len(s))

s = "mIXing CapITALizATIon!!!"
print(s)
print(s.upper())
print(s.lower())

date = "06/28/2022"
print(date.split("/"))

coordinates = ["-1", "3", "-8", "12"]
print("[" + ", ".join(coordinates) + "]")

alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
word = "TOWER"
for c in alphabet:
  if c in word:
    print(c)

print("OWE" in "TOWER")

print("TOWER".find("OWE"))
print("TOWER".find("COW"))
```

Listing 2: String Operations.

3 Dictionaries

The final built-in data structure in Python that we will talk about is the *dictionary*. We've briefly seen dictionaries used before but glossed over their details. A dictionary, also called a *map* in other languages, is a way of storing a collection of data but unlike lists, the data is not ordered linearly. Instead, the data is in correspondence with a set of elements known as *keys* and mathematically, you can think of a dictionary in Python as expressing a bijection between a set of elements known as *keys* and a set of elements known as *values*. Let us take a look at a concrete example of some dictionary usage in Listing 4.

Lines 2 through 7 define a dictionary. You will often split up the definition of dictionaries across multiple lines. The syntax or notation for defining a dictionary are the curly braces `{}` and pairs of key-value elements in between. A key-value pair is of the form `<key>:<value>` and pairs

```python
l = [1,2,3]
n = 3.14159
s = "Hello"
print(f"list = {l}; number = {n}; string = {s}")

n = 3.14159
large_n = 1000000000000000
integer_n = 75
print(f"float with 2 decimal points = {n:.2f}")
print(f"float in scientific notation = {large_n:e}")
print(f"integer in binary = {integer_n:b}")

s = "100.29"
binary_string = "10101"
n = 2022
print(float(s))
print(int(binary_string))
print(int(binary_string, 2))
print(str(n))
print(bin(n))
```

Listing 3: String Formatting.

are separated by `commas`. Observe that you are allowed to have a comma after the very last pair as well. In this example dictionary, the first key-value pair has the key `20` and a value of `"V"`. In general, your keys will usually be numbers or strings. Not all types of elements can be keys. Lists in particular are prohibited and we will see why when delving into how dictionaries are actually implemented.

Lines 4 through 6 define another key-value pair in this example. Here, `key` refers to the variable defined on line 1 and so actually the string `"K"` is used as the key. The value is another dictionary and this nested dictionary has a single key-value pair. Its key is the string `"string_key"` and its value is the string `"string_value"`. To access a specific value in a dictionary, you can index it using its key like on line 9. To get the number of key-value pairs in a dictionary, you can use the `len()` function.

Lines 12 through 17 demonstrate how to add and remove values. To add a new key-value pair to a dictionary, you index into it using the a new key that does not yet exist and assign to the indexed dictionary as if it were a variable. You can also use this notation to reassign the value associated with a key that already exists, as on line 13. To delete a key-value pair, you can use the `del` keyword as on line 16.

Lines 19 through 36 demonstrate how you might use dictionaries in order to represent directed graphs. The nodes are labeled with the letters `"A"`, `"B"`, `"C"`, `"D"` and each key-value pair has a node label as its key and a list of node labels it points to. The graph represented is shown in Fig. 1.

Lines 26 through 36 show three different ways to loop over a dictionary. You can use `.items()`, `.keys()`, `.values()` to loop over the key-value pairs, just the keys, or just the val-

```python
key = "K"
dictionary = {
  20: "V",
  key: {
    "string_key": "string_value",
  },
}
print("Initial dictionary:", dictionary)
print("dictionary[key]:", dictionary[key])
print("Dictionary size:", len(dictionary))

dictionary["another_key"] = [1,2,3]
dictionary[20] = "CHANGED VALUE"
print("Modified dictionary:", dictionary)

del dictionary[20]
print("Dictionary after deletion:", dictionary)

graph = {
  "A": ["B", "C"],
  "B": ["C", "D"],
  "C": [],
  "D": ["A"],
}

print("Graph key-value pairs")
for key, value in graph.items():
  print(key, value)

print("Graph keys")
for key in graph.keys():
  print(key)

print("Graph values")
for value in graph.values():
  print(value)
```

Listing 4: Basic Dictionary Usage.

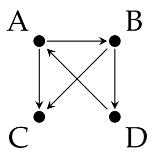

Figure 1: Graph Depicted in Listing 4

ues respectively. These functions return something called view objects and if you want an actual list of the key-value pairs, keys, or values, you can call `list()` on them. For example, `list(graph.keys())` would return the list `["A", "B", "C", "D"]`.

These last few lines of code highlight one useful application of dictionaries. In general, they're useful for storing a collection of data associated with some set of keys, but in particular, they're quite useful for representing directed graphs where the keys are source nodes and values are lists of target nodes of edges. We will discuss graphs and graph algorithms in more detail next week, but let us examine one more simple example of using dictionaries to represent graphs.

Listing 5 demonstrates a few simple operations on directed graphs, namely computing a list of the sink and source nodes respectively in a graph. An intermediary *helper function* we define is `invert()` which takes a directed graph and returns a new directed graph with all the edges reversed. This then makes writing `sources(<graph>)` very easy. It simply returns the the sinks in the inverted graph.

In this example, we are representing directed graphs in the same manner as before. Let us first look at how we implement `invert(<graph>)`. We initialize a new dictionary for the return value and set up key-value pairs for each node in the graph. All the values in the graph are initially empty lists, representing an empty graph with no edges. We then iterate over every key-value pair in the original graph. The `ts` variable in line 5 holds the endpoint of every directed edge starting at `s`. We iterate over these end nodes and appending `s` to `ret[t]` inserts an edge in the opposite direction.

Next, let us take a look at how `sinks(<graph>)` is implemented. Here, we simply iterate over all nodes in the graph and append those without outbound edges. This is easy to check in our representation of the graph, since `len(graph[node])` is precisely the number of outbound edges. Observe that the number of *inbound* edges is not as easy to derive since our representation is asymmetrical. We would have to loop over all the values in each key-value pair in order to compute the number of inbound edges for a given node.

Rather than writing a lengthy `sources(<graph>)` function, we've written a helper function `invert(<graph>)` that may also prove to be useful in its own right. This allowed us to write `sources(<graph>)` simply as a composition of two functions. This illustrates a relatively subtle principle in writing code. Often, judiciously defining and implementing certain "building block" functions will allow you to write other more complicated functions in a simple manner. There isn't really a hard and fast rule of how to decompose your code into separate functions, but if you notice yourself repeating the same type of computation multiple times, that might be a good candidate for putting into a function by itself.

Do exercises 4 and exercise 5, time permitting. Exercise 4 is about checking whether two strings are anagrams of each other and incorporates both strings and dictionaries. Exercise 5 is about computing the transitive closure of a partially ordered set, but uses the same directed graph representation we've discussed.

4 Under the Hood: Numbers and Dictionaries

We will switch gears for a bit to talk about some details of how things in Python work under the hood. Integer and non-integer numbers on computers are generally represented in two different ways. Integer numbers at the lowest level are represented as a series of on and off values or as 0's and 1's. This type of representation is the *binary representation* of a number and writing a number

```python
def invert(graph):
  ret = {}
  for node in graph.keys():
    ret[node] = []
  for s, ts in graph.items():
    for t in ts:
      ret[t].append(s)
  return ret

def sinks(graph):
  ret = []
  for node in graph.keys():
    if len(graph[node]) == 0:
      ret.append(node)
  return ret

def sources(graph):
  return sinks(invert(graph))

example = {
  "A": ["B", "C", "D"],
  "B": ["E"],
  "C": ["F"],
  "D": ["G"],
  "E": ["H"],
  "F": ["H", "I"],
  "G": ["I"],
  "H": [],
  "I": [],
}
print("Graph:", example)
print("Inverted graph:", invert(example))
print("Sinks:", sinks(example))
print("Sources:", sources(example))
```

Listing 5: Simple Directed Graph Example.

in binary is expressing it in base 2 or equivalently as a sum of powers of 2.

An example is $84 = 1 \cdot 64 + 0 \cdot 32 + 1 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 0 \cdot 1$. In binary, we then write 85 as 1010100_2, where the subscript 2 tells us that the number is to be interpreted as a binary string. To compute the binary representation of a number, you can greedily keep subtracting off the largest power of 2 possible until you reach 0. Most programming languages typically use integers of a fixed *bit-width* such as 32 or 64 which is to say that each integer corresponds to a binary string of length 32 or 64, padded with 0's to the left if necessary. Fun fact: when you hear a CPU referred to as 32-bit or 64-bit, it is about the how many bits (short for binary digits) are used to represent a standard integer at the CPU level. Python is a bit unique among programming languages in that handles arbitrarily large integer numbers. They are still represented in as binary strings but there is no cap of 2^{64} or 2^{32}.

Non-integral numbers are also represented as 0's and 1's but follow a different sort of representation. We won't get into the details, but the most common type of representation of non-integral numbers is called *floating-point arithmetic* and non-integral numbers are called *floats*. There is a fixed amount of precision (typically 53 bits worth) and so you will notice precision errors when working with non-integral numbers. Python floats also are subject to the same kind of inaccuracy; in our applications we don't need to worry about precision issues but it's worth being aware of as you might sometimes see output from computations that aren't exactly what you expect but usually only differ by very small amounts. Numerical issues are one reason why computer algebra systems will use more memory in order to represent numbers accurately and compute the desired number of precision when required to do so. One computer algebra system is called `sage` and it builds on top of Python. In the second half of the course, we may cover using Sage since a lot of mathematical objects are implemented and provided by it.

Finally, let us delve into how dictionaries are implemented. Although they are a more complicated data structure, they can be interpreted in terms of lists. The main differences with lists is the method of indexing and the performance of using dictionaries. We have already seen the difference in indexing. In terms of performance, lists can take $\mathcal{O}(n)$ time to insert and remove elements where n is the length of the list. For example, when adding or removing an element from the beginning a list, all subsequent elements will need to be shifted. A dictionary on the other hand, takes on average $\mathcal{O}(1)$ or *constant time* to add or remove elements.

The simplest way you might think of implementing a dictionary is to have a list of all the key-value pairs (each pair is itself a list of 2 elements). However, if you want to find an element given a key or remove an element, you have to potentially look through the entire list each time checking each key. To be more efficient, dictionaries are actually implemented as a fixed size list. Each entry in the list is a "bucket" to hold key-value pairs for the dictionary. Ideally, each bucket would only hold one pair so that you don't have to check all the pairs in the bucket, so this depends on how we assign a key-value pair to a bucket. It should only depend on the key and output a number between 0 and the length of the list minus 1. Such an assignment is given by a *hash function* which takes a key and spits out a number. The hash function should be fairly uniform in that if I have different keys, the likelihood of them getting the same hash is low. This way, buckets generally won't have more than a single key-value pair.

Thus, the improved implementation of a dictionary now looks like: assign key-value pairs to a bucket based on a hash of its key. When a bucket does end up having multiple key-value pairs assigned to it, loop through them to find the correct pair to return or delete. This works well assuming the hash function is not overly expensive to compute and the likelihood of two key-value pairs getting put in the same bucket is low (i.e. the hash function is uniform). Fig. 2 depicts how a dictionary is implemented as a list of buckets, including a scenario where 2 key-value pairs have the same bucket.

However, there is still an issue if the dictionary gets too big! If there are more key-value pairs than the initial fixed size of the list we're using, then any additional pairs added are guaranteed to collide with another pair's bucket. The *load factor* of a dictionary is a ratio of the number of key-value pairs stored to the size of the backing list. As a rule of thumb, if the load factor exceeds 0.75 or so, implementations will create a new list of usually double the size and rehash and store the current key-value pairs in the larger list.

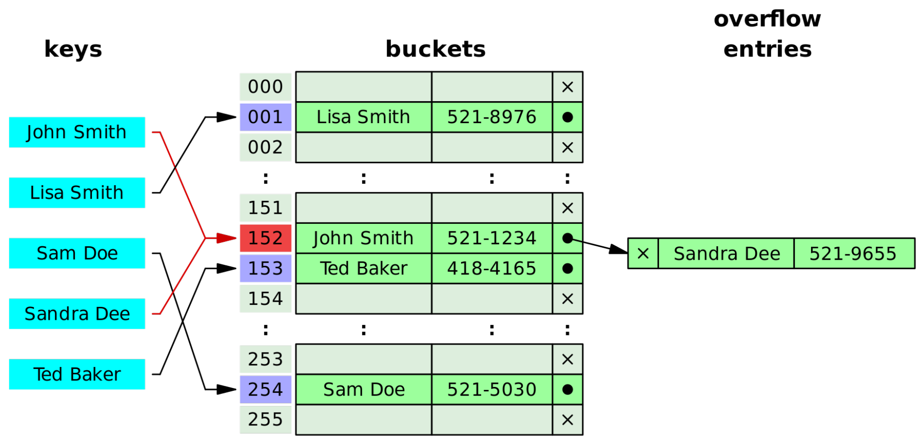

Figure 2: Dictionary Implementation as a List of Buckets

A sample implementation of dictionaries using the ideas we've discussed can be seen in Listing 6. The actual implementation used in Python is quite complicated and takes up over 5000 lines of code! You can see the code on GitHub However, the general ideas are the same and a lot of the additional complexity comes from providing additional built-in dictionary functionality, handling various edges cases, and iterating through buckets with multiple key-value pairs differently. Look for the `USABLE_FRACTION` value to see that the actual load factor used in practice is set to $\frac{2}{3}$.

```python
LOAD_FACTOR = 0.75
ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

def new_dictionary(size=4):
  store = []
  for i in range(size):
    store.append([])
  return [store, 0]

def hash(modulo, string):
  ret = 0
  for i in range(len(string)):
    ret += ALPHABET.find(string[i]) * (26 ** i)
  return ret % modulo

def get(dictionary, key):
  store, _ = dictionary
  h = hash(len(store), key)
  for k, v in store[h]:
    if k == key:
      return v

def add(dictionary, key, value):
  store, _ = dictionary
  h = hash(len(store), key)
  store[h].append([key, value])
  dictionary[1] += 1

  if dictionary[1] / len(store) > LOAD_FACTOR:
    resize(dictionary)

def resize(dictionary):
  print("Resizing...")
  store, num_entries = dictionary
  store_ = []
  for i in range(2*num_entries):
    store_.append([])
  for i in range(len(store)):
    for key, value in store[i]:
      h = hash(len(store), key)
      store_[h].append([key, value])
  dictionary[0] = store_
```

Listing 6: Sample Dictionary Implementation.