Python Session 2

After going through the content, please take a minute to respond to this feedback survey of our reference manuals. Feel free to use the outline on the left side of the document to jump to different sections.

Lists

Creating Lists

Lists allow us to store multiple items in a variable. Naming conventions for lists follow the same rules as variables. Each item in a list is called an element. List elements are wrapped around two brackets **`[ ]`** in which a comma separates each item. Unlike *C++*, Python allows you to print an entire list just by passing it into the **`print()`** function.

```
list1 = [1, 2, 3, 4, 5]
print(list1)
```

```
[1, 2, 3, 4, 5]
```

Lists can also be created using multiplication to repeat the list items. To do so, add a **`* <number_of_duplicates>`** after the list, and it will repeat the whole list by number_of_duplicates times:

```
list1 = [1] * 3               # list1: [1, 1, 1]
list2 = [1, 2, 3] * 2         # list2: [1, 2, 3, 1, 2, 3]
```

Unlike *C++* and many other languages, Python allows lists to have a mix of data types. Lists can also have more lists stored within them, which will create a **2D list**.

```
list1 = [1, True, "Galileo"]                  # mixed list
list2 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]     # 2D list
```

List Iteration

Once we've created a list, we may want to use them. We can **iterate** through a list to access each **individual** item in the list by using a **`for`** loop:

```
list1 = [1, 2, 3]
for i in list1:
    print(i)
```

```
1 2 3
```

In this **`for`** loop, the **`i`** stores the value of a single item each time. As it loops, it goes to the next item until the end of the list is reached. This is similar to **`for (auto i : list1) {}`** in *C++*.

Thought Experiment [†]

How can we iterate through a 2D list?

Without loops, we can also get list elements by using the brackets **`[ ]`**. We can pass in an **index** or a **range** to get a single or sub-list of items from the existing list. Note that indices start from **`0`** and go up to **`list_size - 1`**. However, unlike *C++*, negative indices (**`-1`** to **`-list_size`**) can be used to iterate from the back of the list instead.
There are a few ways to get elements with brackets **`[ ]`**:

- **`list1[0]`** - get the element at **index 0** (first element)
- **`list1[1:3]`** - get element from **index 1 to 2, *excluding 3***
- **`list1[:3]`** - get element from ***beginning*** **of list to index 2, *excluding 3***

- `list1[3:]` - get element from **index 3 to *end* of list**

The colon allows you to get a sub-list of elements in a range you provide.

```
list1 = [1, 2, 3, 4, 5]
print(list1[0])             # gets element at index 0          Outputs: 1
print(list1[1:3])           # gets sublist from index 1 - 2    Outputs: [2, 3]
print(list1[:3])            # gets sublist from index 0 - 2    Outputs: [1, 2, 3]
print(list1[3:])            # gets sublist from index 3 - end  Outputs: [4, 5]
```

Modifying Lists

Sometimes, we may want to modify the contents within a list. We can edit an item's value by using the same bracket iteration method mentioned above.

```
list1[0] = 18           # modifies the first item's value to 18
```

If we want to add or remove elements from the list, we can use the list's built-in functions:

- `<list>.append(x)` - add the item **x** to the **back** of the list
- `<list>.insert(index, x)` - insert item **x** into the list at the **index** provided
- `<list>.pop()` - removes the **last** element in the list
- `<list>.remove(x)` - removes the first item **x** provided

```
list1 = []
list1.append(3)              # list1 = [3]
list1.append(10)             # list1 = [3, 10]
list1.insert(0, "Hello")     # list1 = ["Hello", 3, 10]
list1.pop()                  # list1 = ["Hello", 3]
print(list1)
```

```
['Hello', 3]
```

Dictionaries

Dictionaries are extremely efficient data structures and behave similarly to how actual dictionaries work. Rather than indices, they use a **key** to hash to an index that will store a value. Dictionaries are created by wrapping its elements around **{ }** with the following syntax: **`{"key1":value1, "key2":value2, ... , "keyN":valueN}`**. Elements are accessed using **[ ]**, with a single key passed into it. Dictionaries are similar to **`map<dt_1, dt_2> var`** in *C++*.

```
dict1 = {"A":65, "b":98, "Scott":"Scooter"}
print(dict1["b"])
print(dict1["Scott"])
```

```
98
Scooter
```

Dictionaries can also be created from 2 parallel arrays. These arrays must be **zipped** before creating the dictionary.

- `zip(a, b)` - zips two arrays together so that each element is a **pair** with the other element at that index
- `dict()` - creates a dictionary given the passed in list of **pairs** of keys and values

```
names = ["Jason", "Kenneth", "Andrew"]
colors = ["Red", "Green", "Purple"]
fav_colors = dict(zip(names, colors))   # uses names as keys and colors as the values
print(fav_colors)
```

```
{'Jason': 'Red', 'Kenneth': 'Green', 'Andrew': 'Purple'}
```

⚠ Make sure the length of the lists are the same to properly pair the keys and values together.

Existing entries can be modified by **indexing the key** and assigning a new value to it. Likewise, a *new* entry can be appended to the dictionary by indexing a new desired key and assigning a value to it. Python will automatically detect that this key does not exist and will add the new entry to the dictionary.

```
names = ["Jason", "Kenneth", "Andrew"]
colors = ["Red", "Green", "Purple"]
fav_colors = dict(zip(names, colors))
fav_colors["Jason"] = "Orange"                # modifies an existing entry
fav_colors["Turing"] = "Black"                # creates new entry at the end
print(fav_colors)
```

```
{'Jason': 'Orange', 'Kenneth': 'Green', 'Andrew': 'Purple', 'Turing': 'Black'}
```

Strings

Strings are useful for a lot of applications. But what exactly are they? Strings are the same as a list of single characters. This means that we can iterate through strings to get **substrings**. Indexing a string follows the same rules as indexing a list.

```
word = "supercalifragilisticexpialidocious"
substr = word[:5]      # gets a substring from index 0 to 4
print(substr)
print(word[-7:])       # substring from the seventh to last letter (d) to the end
```

```
super
docious
```

Format Strings

Last session, we learned how to combine strings with commas and pluses. This could get messy or cluttered, especially with concatenations. Luckily, Python also allows for **format, or f, strings**. F strings are **simple and more readable** as all we need to do to combine the values with the string is to insert them into the string with **{ }**.

⚠ The strings are not f strings by default! We must tell python that we're creating an f string by prepending the letter **f** outside of the string.

```
print(f"I have {7} cookies in the fridge")    # prepending f creates an f string
```

```
I have 7 cookies in the fridge
```

We can have multiple of these within the string, and we can also pass in variables. These f strings can also be stored in a variable to be used later.

```
num_comps = 4
name = "Sahas"
phrase = f"{name} has {num_comps} computers!"   # we can store our f string
print(phrase)            # prints out the f string with everything in it
```

```
Sahas has 4 computers!
```

String Functions

There are many useful functions that allow us to modify or look through strings. *Most* of these string functions will be called (used) in the following format: `<string>.<function()>`. Unlike many other languages, these functions can also be **called on string literals**!

The following functions will **return** a string with the changed cases of the characters within a string. These functions do not change the actual string and only **returns** a modified copy of the string.

- `upper()` **-** returns a string with all characters **uppercased**
- `lower()` **-** returns a string with all characters **lowercased**
- `swapcase()` **-** returns a string with all the characters' cases **swapped (lower → upper, upper → lower)**

```
command = "put the cookie down!"
print(command.upper())
```

```
PUT THE COOKIE DOWN!
```

`find(x)` will **return** the index of the **first instance** of x found in the string. x can be a character or a substring. If no index is found, the function will return `-1`.

⚠ `.find()` is **case sensitive**! If the string is in the wrong case, it may return a `-1`!

```
a = "Link's shield"
print(a.find("i"))                  # finds the FIRST i in the string
print(a.find("link"))               # finds the substring "link"
print(a.lower().find("LINK".lower()))  # finds the substring regardless of case
```

```
1
-1
0
```

Thought Experiment [†]

In the above example, the last `print()` is able to find the provided substring in the string regardless of the case of **both** the string and the passed in substr.
How exactly does the code successfully find the substring?

The following functions will **return** a string given its specific modifications. These functions do not change the actual string and only **returns** a modified copy of the string.

- `reversed(x)` **- returns** the reversed version of the string that is passed in. The format to use this function is: `list(reversed(<string>))`. We need to use `list()` to make the output readable as a list. This function can also be used to reverse a list.
- `strip(x)` **- returns** a version of the string with the specified characters removed from both ends. We can pass in a character, string, or leave it empty (spaces). If a character is not at the end at some point, it is not stripped.
  - `lstrip(x)` removes from only the left end; `rstrip(x)` for the right end
- `replace(old, new, y)` **- returns** the string with `y` amount of `old` substr/char replaced with the `new` substr/char. If no value is passed to `y`, it is defaulted to replace **all occurrences** of `old`.
- `join([x])` **- returns** a string with the list of strings joined by the string from which the function is called.

```
a = "Hornet"
print(list(reversed(a)))     # returns reversed a string as a list

empty = "  EMPTY  "
stripped = empty.strip()          # default: strips spaces from the ends of the string
print(stripped)                   # stripped = EMPTY
print(stripped.strip("EPY"))      # strips the ends of "EPY". "P" is not at the ends

word = "racecar"
print(word.replace("a", " ", 1))      # replaces 1 (the first) "a" with " "

print("-".join(["Python", "Session", "1"])) # each string is joined by a "-"
```

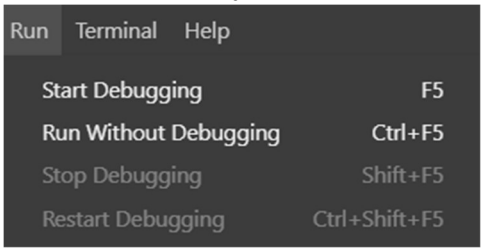
```
['t', 'e', 'n', 'r', 'o', 'H']
EMPTY
MPT
r cecar
Python-Session-1
```

Extras

Compiling/Running Recap

Here, we will provide a quick recap of how to compile and run our python programs. Unlike many other programming languages, we only need to type in a single command in our terminal to both compile and run our python scripts!

`python3 [filename]` will compile and run your script. Remember to include the **`.py`** extension!
You can also run your code with the **Run Without Debugging** button at the top under **Run** or **Ctrl+F5**:



C-like f Strings

This session, we introduced the use of **f strings**. For people who are experienced with *C* or *C++*, you may have heard of this before, or may have seen a variation of this. Specifically, we are talking about formatting strings with **format specifiers**. This exists in Python too!

```
print("Hello %s, Age: %d!" % ("Andrew", 20))
```

```
Hello Andrew, Age: 20!
```

In our strings, we can specify that we are using c-like format strings by adding a `%` after the string. Note that **`f`** is **not** prepended here. After the `%`, we can put our format inputs in order within the parentheses. These inputs will replace each `%*` we put inside our string in the order you provide them. This is different from the **f strings** learned in this session as we must correctly specify the type it will be.
For those unfamiliar with this format, we suggest using the method that was taught this session; however, if you wish to empower yourself with this knowledge, you can learn more here.

Runtime in a Nutshell

Computers seem magical, but they are actually very limiting! They do exactly what we tell them by using up resources that we provide for them. Each action takes a certain amount of time to perform. But, why is this important to know? Although a computer only takes a few milliseconds to perform each operation, these can build up. For your understanding, we will scale up the time for **`print()`** to 1 second.
A single print statement will use a second. But what happens if we put this in a loop? It will cost **1 second * how many times we loop**. If we nest the loops, this gets even worse:

```
...
for i in list2:
    for j in list2:
        print(j)
```

Each outer **`for`** loop will repeat some amount of times. For each of those repeats, the inside **`for`** loop will also run and repeat some amount of times. For example, if the inner **`for`** loop repeats 5 times, each repeat of the outer **`for`** loop will run the inner **`for`** loop again, which will print 5 times for each run of the outer loop. If the outer **`for`** loop repeats 10 times, there will be a total of **5 * 10 = 50 prints**, costing **50 seconds!**

The more loops there are, particularly nested loops, the slower the program will become. Even more, in more complex algorithms where the number of repeats is unknown or decided by the user and may repeat **n** times. Mathematically, nesting **k** loops that repeat **n** times will have us repeat operations **n * n *... * n_k = n^k times**, meaning an **exponential growth in time!** If you would like a more in-depth/computer-science explanation of this, we suggest looking up runtime complexities or Big-O.

Thought Experiment Answers

TE1: We can iterate through a 2D list by using 2 for loops, one nested in the other

```
list2 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]       # 2D list
for i in list2:
    for j in i:
        print(j, end=" ")
    print("")
```

```
1 2 3
4 5 6
7 8 9
```

TE2: **`print(a.lower().find("LINK".lower()))`** finds the characters regardless of the character's case. It does this by getting the lowercase version of the string **a**, then using **.find()** on that string. It then uses the lowercase string returned by **"LINK".lower()** to search with.
The process looks like this:

`a.lower() → "link's shield" →`
`"LINK".lower() → "link" →`
`→ "link's shield".find("link") → 0`