Agenda

1. Administrative
2. Review
3. String slicing
4. String methods
5. Lists
6. Mutability

Administrative

We did a poll of how long checkoff 2 took that also has an opportunity for providing feedback. If you'd like to add anything, the form is still open here: www.bitly.com/6149-checkoff2

Poll results as of 1/14/2015, 7 p.m.

- Most people took 30-90 minutes to complete checkoff 2. If you're spending more time than that on the checkoffs, **do your work in the office hour room so that you can add your name to the help queue when you get stuck.** Asking for helping is encouraged - nobody enjoys banging their head against a wall. Ten minutes is a good amount of time to spend trying to fix a problem yourself.
- As always, some people think the pace is too fast - especially over more challenging concepts - while others would prefer that we spend less time covering the same topics as the readings. Our goal is to make sure everyone leaves this class know how to code, so we do intend to err on the slower side of things - we want everyone to have a passing grade!
- We will provide practice problems on the course website; those of you who have been doing the readings and find lecture boring should still come to class, but when you get bored, focus your attention on these practice problems. (We still want you in class so you can learn "tricks of the trade" and for the sake of a structured learning environment.)
- Instead of starting class with review, we'll start with the newer, more challenging topics to make sure they get enough class time. Don't worry, we're getting to the point where we have enough tools to tackle more interesting problems!
- Checkoffs queues take a long time. We know, and we warned you that you needed to ensure adequate time in your schedule to complete checkoffs. We've also explained that in-person checkoffs are important for maintaining academic integrity and checking your conceptual understanding of the material. Unfortunately, our checkoff procedure won't be changing. **We will be adding more LA support to morning office hours, but please come to office hours 2-4 p.m. if you can, as those times are more empty.**

- There are a couple vague criticisms of the instruction (e.g., “The instructor isn't very clear about certain important knowledge.”), **but nobody is asking questions in class**. If you have questions or think a topic needs to be emphasized further, it is *your* responsibility to raise your hand and ask. Additionally, Michelle personally appreciates constructive feedback - if there’s specific changes that would make your time in lecture more useful that aren’t mentioned here, PLEASE shoot her an email (mszucs@mit.edu)!
- **Nobody said the readings aren’t helpful!** We suggest doing the readings if you aren’t already.

Checkoff 3 will take much longer than Checkoff 2 (maybe twice as much time). Ask questions on Piazza and come to office hours on Thursday. Talk to the instructors if you find yourself spending more than 2-3 hours on this assignment, and we can get you some individualized help and/or post hints to the class as a whole.

In general checkoffs will be uploaded around when the previous one is due. We’ll try to do it earlier so people can work ahead, but no guarantees.

Checkoffs can happen early! Work in office hours! We’re putting together a bank of advanced problem solving activities for those of you who want to spend more time writing code.

It’s really important to read emails from the class, read the syllabus, etc. Michelle tries to not spam people, so if you’re getting an email, it’s important.

Review

1. Instead of typing:

   ```
   number = number + 1
   ```

   you can type:

   ```
   number += 1
   ```

   -=, /=, and */ are also valid operators.
2. Since checkoff 3.1 is harder, you should write pseudocode - or an English explanation of what you plan to do - before you write Python code. When there’s multiple approaches to a problem, it really helps to write down the path you’re planning to take so you’re forced to commit to one method. Coding is basically building a logical bridge from a set of inputs to a desired output - it’s a lot easier to think in English (or your native language) than it is to think in a programming language you’ve only known for 3 days.

3. Review nested if/else statements from the lecture 2 notes - there's a slightly longer example we didn't cover in class, along with some tips for how you can combine if/elif/else.

String slicing/index basics

Python knows how long a string is. To get the number of characters in a string, use the *len* function.

```
pet = “cat”
print len(pet)
```

The above code outputs 3. Python can also tell you what the character at a specific position in a list is. We call each position an *index,* and Python is *zero-indexed*, which means that the first value in the string is said to be at index 0, the second at index 1, and so on.

To a particular element of the list, we use square braces. For example:

```
pet = “cat”
print pet[0]
print pet[1]
print pet[2]
```

outputs

```
c
a
t
```

Placing a number equal to or greater than the length of the string in the braces will result in an error.

We can also access *slices* of strings, or series of consecutive characters. The general format for accessing a slice of string `my_string` is:
`my_string[a:b]`
where `a` is the inclusive start of the substring and `b` is exclusive end of the substring.

Both `a` and `b` are optional. `a` defaults to 0, and `b` defaults to `len(my_string)`. Slicing returns a **copy** of the subsection of the string - i.e., it does not change the original string. If you want to alter the original string, you need to do variable assignment as we've seen throughout the course. Try the following code in your shell.

```
pet = “cat”
print pet[0:3]
print pet[1:2]
```

String methods

There are a number of special functions that only work on strings. When functions are associated with one particular type, or class, they are called methods. We call (remember, *to call* means *to use*) these special methods in a manner a little different from the other functions we’ve used.

The methods that are reserved for strings don’t all load automatically when you open your Python shell. Instead, we need to ask Python to let us use these methods. If you’re working in a saved .py file, type the following line of code at the top; if you’re in the shell, type it and hit enter:

```
import string
```

Not surprisingly, “string” is the name of the *library* of code that contains extra string methods for us.

Now that we have access to the code, let’s look at how we call (or use) these methods. Remember that previous functions we’ve use have been in the format `name_of_function(input)`. In this class, we’ve done `len(pet)`, where `len` is the name of the function and `pet` is the input to the function.

Methods are a little different - we use *dot notation*, which is of the format:

```
primary_input.function_name(additional_inputs)
```

`primary_input` must always be of the type that the method applies to. When we’re working with string methods, `primary_input` should always be a string. Let’s check out the `lower` method, which returns as output a copy of the input string, with all lowercase characters.

```
greeting = “Hello”
print greeting.lower()
```

Remember how expressions get evaluated in Python when figuring out what happens in the second line of code. greeting is evaluated to “Hello”; lower() is called with “Hello” as input; then the result of the function call is passed as an argument to print.

Other commonly used string methods are `find` and `count`. Rather than spending a long time going over the boring details of these functions in class, we'll direct you to the Python documentation: https://docs.python.org/2/library/stdtypes.html#string-methods

One last note about strings. You can have an *empty string* by just typing a set of quotation marks that does contain characters. Try the following code:

```
if “”:
      print “Empty strings evaluate to True.”
if “any string that isn’t empty works here”:
      print “Non-empty strings evaluate to True.”
if 0:
      print “0 evaluates to True.”
if 4:
      print “Any non-zero number evaluates to True.”
if 2 == 3 or 4:
      print “This doesn’t mean if 2 is equal to 3 or 2 is equal to 4.”
```

Which lines print? How does this fit into your rock-paper-scissors code? Ask any questions you have on Piazza.

Lists

Lists are exactly what they sound like - collections of objects. In Python, lists can store any other object; they can be comprised of only one type of object, or objects of many different types. Lists can even contain lists!

An empty list is created using square braces. You can also add elements to a list right away by including the elements, separated by commas, between the brackets.

```
empty_list = []
non_empty_list = [1, 2, 3]
```

Indexing to retrieve elements and slices works identically to strings.

If you have nested lists, you access elements using two sets of square brackets of the form my_list[3][2]. The expression is evaluated from left to right, so my_list[3] becomes the list stored as the 4th element of my_list, then the 3rd element of that sub-list is found.

Lists are what we call mutable, which means they can be changed after they are created. That's not true of strings, or any of the other types we've worked with so far - the only way we "change" strings so far is through reassignment, as follows:

```
string = "Original string"
string += " and some extra stuff"
```

That's the same as

```
string = string + " and some extra stuff"
```

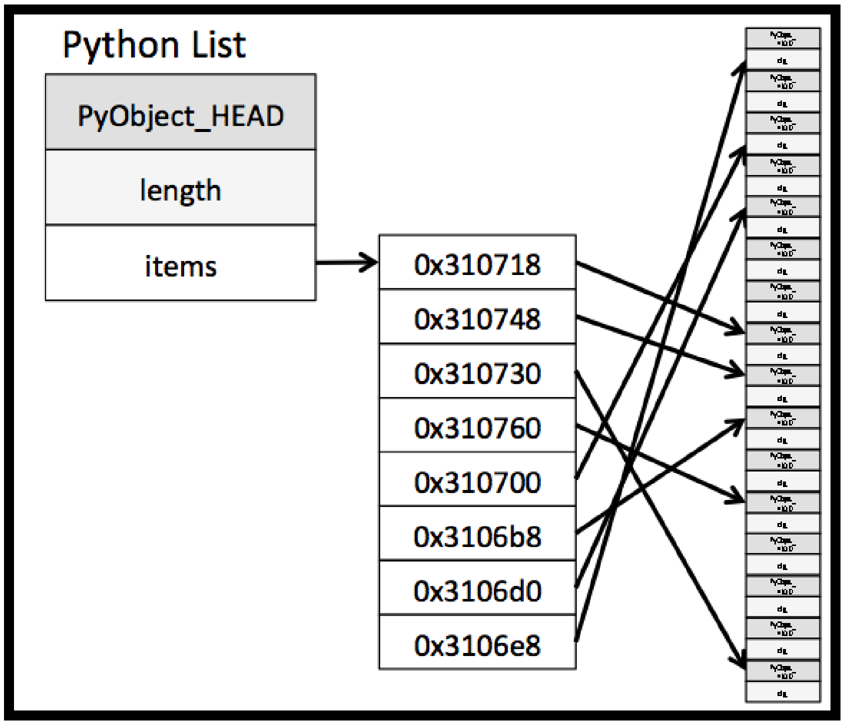
The above code is no different than saying `string = 3`; the variable is being completely unlocked from it's original value and set to a new value.

With lists, the following code is valid. Change changed_list to a string and try to reassign the second character to convince yourself it doesn't work.

```
changed_list = [1, 2, 3, 4]
changed_list[1] = 8
print changed_list
```

Michelle's note: I think the following explanation of why lists are mutable is actually really bad, and I'm trying to figure out a better way to explain it.

Why are lists mutable? This picture (taken from https://jakevdp.github.io/blog/2014/05/09/why-python-is-slow/) may help you understand how Python stores lists.



All the crazy numbers represent segments of memory in your computer. A variable name representing a Python list is associated with a block of memory that stores the locations of *other* blocks of memory. If two list are set equal to each other, they simply point to the same block of memory - two strings would have the same values stored in *different* locations.

When we set two lists equal to each other and they point to the same object, it's called *aliasing*. Run the following block of code to see that the two lists change together.

```
list_1 = [1, 2, 3]
apparently_list_1 = list_1
list_1[1] = "Evil"
print list_1
print apparently_list_1
```

If you're not careful, you can *really* mess up your code with aliasing and mutation issues. We'll see more example throughout the course.

To avoid this type of problem, you can easily assign b to a *copy* of `list_1` using splicing. Try the following block of code:to avoid?

```
list_1 = [1, 2, 3]
definitely_not_list_1 = list_1[:] # This is the line that's different!
list_1[1] = "Evil"
```

```
print list_1
print definitely_not_list_1
```