Agenda

1. Review: Variable names
2. Review: Strings
3. Calling print with multiple inputs and string manipulation
4. User input
5. Booleans and comparisons
6. If/elif/else statements and control flow

Update: The checkoff 2 file has been updated as of 10:18 a.m on 1/13/15. Make sure you downloaded checkoff2_updated.pdf! Clear your cookies/cache if you don't see this file on the course website.

Review: Variable names

Variable names can contain (but not start with) numbers. Variable names can contain (and start with) underscores.

By convention, variable names starting with underscores are reserved for special cases - you probably won't start your variable names with underscores in this class.

When making descriptive variable names, you'll often need to string a few words together. We do so by putting underscores between the word; this format is called snake case. As we progress through the course, we'll use other formats for the names of functions and classes we create.

For now, start your variable names with lower case letters.

Review: Strings

Recall that to tell Python to store a value as plaintext, or a string, we enclose the value in quotation marks. You can use single or double quotation marks, which helps in saving strings that include quotation marks and apostrophes. Compare the following attempts at creating a string:

```
bad_sentence = 'Mary's lamb ran away.'
good_sentence = "Mary's lamb ran away."
```

`bad_sentence` throws an error because the apostrophe in "Mary's" terminates the string early, resulting in a close quotation mark without an opening match.

Remember what we said yesterday about the output of using an operator being dependent on the types of the values in the expression. In addition to being able to multiply integers, you can also “multiply” strings.

```
greeting = “hello”
hello_hello =  greeting*2
```

`hello_hello` has the value “hellohello”.

Calling print with multiple inputs and string manipulation

Some functions can take a variable number of arguments, or inputs. Print is one of those functions. To pass in multiple values as input to a function, separate the arguments by commas. When printed, the arguments will automatically be separated by spaces.

```
part_1 = “Hello”
part_2 = “my”
part_3 = “name”
part_4 = “is”
part_5 = “Michelle”

print part_1, part_2, part_3, part_4, part_5
```

prints “Hello my name is Michelle”

We usually see commas after “Hello” in these types of sentences. Remember that Python evaluates each term individually before passing it on as input to print. We can add additional information to the first term by using the following call to print instead:

```
print part_1 + “,”, part_2, part_3, part_4, part_5
```

prints “Hello, my name is Michelle”. Note that we still don’t need to include a space after the comma we add to “Hello”; the entire unit is still evaluated to a single argument that’s sent to the print function.

User input

There are two Python functions that you can use to get information from an interaction with the user. The first, which we will use for this class, is `raw_input`. The second, which you should not use, is `input`.[1]

lec02.py shows an example of using `raw_input` to get user input. `raw_input` takes one[2] argument. The argument should be a string and tell the user what to type, as it will be printed to the screen.

Tip: Including a space at the end of the string prevents the user's input from being crowded on top of the prompt when the code is run.

Booleans and comparisons

The next type we'll examine in Python is the *boolean* type. While a float can take on infinitely many values of numbers, a boolean variable can only take on one of two values: *True* or *False.*

The primary use of boolean variables is in performing comparisons that cause certain blocks of code to be executed only if certain conditions are met. Before we examine the concept of control flow, let's look at these comparisons and some special keywords.

Boolean expressions - like all expressions in Python - are evaluated. Boolean expressions are repeatedly evaluated until they boil down to either True or False.

We can compare whether one value equals another (==), is not equal to another (!=), is greater than (>), less than (<), greater than or equal to (>=), or less than or equal to (<=) another number. In your shell, try:

```
2 == 2
```

When you hit enter, the shell will display True, because 2 and 2 are obviously equal to each other. Try other comparisons are well. You can also put expressions on either side of the comparison operator:

```
2 + 2 == 4
```

prints True.

---

[1] `input` evaluates whatever the user types as Python code; a malicious user could type code that, for example, deletes your file system. Because of this security vulnerability, we do not want to encourage you to use `input`.

[2] You can actually call `raw_input` without any arguments, but the user won't be presented with a textual prompt, which makes it hard to know what to type!

If/elif/else statements and control flow

Remember that Python interprets code line by line. So far, every line of code that we've written has been executed (unless there's error that prevent the code from running). In some cases, we might want to set aside certain blocks of code that are only executed if certain conditions are met.

For instance, when you pay for items on campus with TechCash, you're able to set a low balance warning. The cash registers display a notice that your balance is low if - and only if - you have less money in your account than the threshold you set.

We use (aptly named) *if statements* to section off portions of code. Following the word "if", you will write a boolean statement, followed by a colon. If that statement evaluates to True, the code inside the if statement is executed; if the statement evaluates to False, the code is ignored. Code "inside" the if statement is denoted by indentation; any lines indented one level after the if statement are associated together, until a de-dented line is typed. Let's examine the following code, assuming we have a variable named *balance.*

```
if balance < 5:
      print "Your account balance is low."
print "Thank you for using TechCash."
```

The first print statement is inside the print statement (and therefore only executed when balance is less than 5). The second print statement is outside the if statement, and therefore always executed, regardless of the value of balance.

An *else* statement can follow an if statement, and will be executed when the boolean in the if statement evaluates to False.

```
if balance < 5:
      print "Your account balance is low."
else:
      print "Your have more than $5 in your account."
print "Thank you for using TechCash."
```

Else statements can include additional if statements. You will need to indent your code inside the second if statement another level.

```
if balance < 5:
      print "Your account balance is very low."
else:
```

```
      if balance < 25:
            print “Your balance is getting low.”
      else:
            print “You have more than $25 in your account.”
print “Thank you for using TechCash.”
```

Since that’s a little clunky to write, we can instead condense the code using the combo *elif* statement.

```
if balance < 5:
      print “Your account balance is very low.”
elif balance < 25:
      print “Your balance is getting low.”
else:
      print “You have more than $25 in your account.”
print “Thank you for using TechCash.”
```

Notice the difference in indentation.

Hints:

- if can be used without elif or else
- elif can only be used after an if; can be followed by other elifs; and can be followed by 0 or 1 else statements
- else can only be used after an if statement (with 0 or more elifs betweens the if and this else); and terminates the if-else logic block (i.e., any subsequent elifs must be preceded by a new if, and any subsequent ifs will be unrelated)