Test 1 Review Guide CS 112

Bring your GMU ID and a pencil to the test.

This document describes info and how to study for your upcoming tests. It is split into three sections for each test, but we might need to modify the stopping points or topic ordering for each test as the semester progresses.

This is a somewhat overly detailed description of the different things we've learned and explored this semester. If you'd prefer a shorter laundry-list of topics, just read the section headers to get an idea of what topics we'll cover; the extra detail is to give more information when you want it, but we're not saying you're supposed to be memorizing this document at all! It's about performing programming, not rote memorization. The tests will spend some time testing your knowledge of definitions and terminology, but will primarily be interested in your ability to either look at code samples and show understanding of what happens, or to write small code samples that achieve a simple effect. The test will be roughly 50%/50% split between short-answer questions and applied coding questions. The first portion will be some theory/definitions, but will also attempt to test concepts by asking questions such as "which option matches the correct printout of the following code?", or "what is printed by running the following code?".

How to prepare:

- Look to the "Practice Problem" slides **and TopHat questions** for excellent ideas of what small questions may be on the test.
- Lab materials are excellent samples of what might be asked on the tests. Lab quizzes are like single pages from tests, and lab tasks are like sample test coding questions.
- Many of the examples listed below (+EX) were in the slides; review them.
- You should also spend some time looking through the examples we've been modifying in class. They are available as zip files on Piazza, as well as some auxiliary problems also uploaded. Ask for help if downloading them and being able to edit/run them doesn't work for you.
- If you only read the text and try to memorize stuff from slides, you will definitely **not** be prepared for this test: practice writing programs, practice coding! That's a major focus; this simply isn't a memorization test.
- Everything on our test will assume Python 3 code.

Python Interpreter

- "chat window" between you and Python
  - type python (PC) or python3 (mac) at a terminal/command window.
  - the >>> symbol means Python is ready for you to type expressions and commands.
- give it work and it does it for you.
  - expressions: interpreter reduces to value, reports back (by representing the value on the next line)
  - statements: does the command, is then ready for more
- contrast with module/source mode, where Python simply executes all the lines in a file and then can quit immediately (never offering the >>> prompt for us to type in extra values—the program does exactly what the lines of code state, and no more)
- type python sourcefile.py (or python3 sourcefile.py for macs) at terminal/command window.
  - Python opens the file, executes all the code in it, and once it finishes, Python quits.

Numbers

- Python uses plain old decimal numbers (base 10). These are called ints.
- we also can *approximate* real numbers (numbers with a part after the decimal point). These are called floats.
- difference between integers (ints) and reals (floats): understand when python tries to use one versus the other.

Expressions

- representations of values. E.g.: 3+4 "hi" True
  2+(3*int(input("num plz:")))
  - can then be 'reduced' down to some value (7 in the first example above).
- expressions can include lots of things that we can reduce to a value: function calls, operators, sub-expressions, variables, literal values, etc.
- some math operators: + - * / // % **
  +EX: example usage of each in chart.
  - Precedence: Python obeys PEMDAS. There are many other operators that also have related precedences, so that we can skip some parentheses if we also remember the order of precedence. When in doubt, add enough parentheses to be sure what happens first.
    + EX: precedence chart.
    + EX: large expression; show each step of evaluation.
    + EX: use Python's precedence to evaluate some mathematical expressions.
- variables: variables are names that refer to spots in memory. When we see a variable in an expression, Python looks up its current value in memory and 'reduces' the variable to its current value.
  + EX: >>>x=5 # let's create the variable first, so we can use it soon.
  → Assignment statement. If the variable x didn't already exist, python obtains a spot in memory for x to refer to. Python stores the value 5 at that memory location.
  >>> x+4
  → Expression. Python looks up x, finds out that its value is 5, and then simplifies the expression down to 5+4.
  → Python continues, reducing 5+4 to 9. Since Python is done with reducing the expression, and we're in interactive mode, it prints 9 back to us so we can see the result.
- function calls: lets us use code written elsewhere by others (and later ourselves)
  - We must always have the trailing open/close parentheses, even when there are no arguments.
  - just like math expressions, they are 'simplified': simplify any argument expressions to values, then run the code in the function, resulting in a value (returning with a value).

Variables / Assignment

- when we want to store values for later.
    - don't have to write a single, large, complex expression
- by giving a new name and ASSIGNING a value to it, Python takes care of choosing a spot in memory, storing the value, and recalling the correct value for us when we ask for it later on.
- assigning a value to a variable:
  variableName = expression
    - get the value of the expression on the right, and store it into the variable named variableName.
    - read it as "assign the value of this expression to the variable named 'variableName'".
        + EX: assign some values; then use them in an expression.
    - can ONLY have a memory location (like a variable name) on the left of =; and can have any expression on the right-hand side.
        + EX: some valid and illegal assignment statements
    - = is not like in math! Not a statement of equality, but a command to assign a value to memory.
        + EX: x = x+1 is valid code. It increments x (x now stores a number one greater than before).
- TIME flows ever forward--all we ever know about a variable is its *current* value. We can update (replace) its value with another assignment statement.
  + EX: x=5; y=10; print(x); print(y); x=y+3; print(x); print(y);
- Create a variable by assigning to a variable the first time
- Programming TRAP: if you misspell a variable name when assigning to it, Python treats this as a new variable creation, and happily/obediently creates the quite-similarly-named variable. Your code will look right but perform oddly until you notice the typo!
- "Augmented" Assignment: +=, -=, *=, etc.
    - x += expr means x = x + (expr)
    - note the parentheses in the expanded version!

Identifiers

- the names we create and use in Python. Used for variables, function names, modules
- Rules: start with a letter or underscore. Can continue indefinitely with any letters, digits, or underscores.
- keywords are identifiers that already have meaning in Python; we can't re-use them for our own identifiers.
  +EX: chart of Python keywords.
- Built-in stuff: Python already has lots of functions and things defined--each has its own identifier.
    - e.g.: input(), int(), str()...
    - you can actually redefine these identifiers. But it's a bad idea to reuse these names
- user-defined: as you create your own variables, you are creating new (user-defined) identifiers.
- convention: (a great idea, but Python doesn't enforce this)
    - use identifiers that are descriptive or meaningful for their usage.
        + EX: current_age instead of a
    - this helps make your code 'self-documenting'.
        + EX: self-documenting code example. (gallons-to-mpg)

    + EX: lots of identifiers; which ones are valid and well-named? Which are valid, but poorly chosen? Which ones aren't even allowed by Python?
- case (upper/lower) is significant.
  + EX: all these identifiers are different/distinct: myvar MyVar myVar my_var myVar1 MyVaR

Statements

- a command for Python to do something.
- different than an expression. While Python will reduce expressions down to values for us, an expression doesn't say what to do with it. (In the interactive interpreter, Python assumes we want to see the results so expressions' values are printed for us. This is a special case).
- Assignment (e.g., x = 2 + 3) is a statement: "reduce this expression on the righthand side to a value, and STORE IT to the variable with this name on the lefthand side."
- compound statements: grouping multiple statements into one unit. Used in functions and control structures when we create an indented block of code.
    - control-flow statements are compound.

Input/Output

- lets code be more general
    - simplest input: ask user to type input
        + EX: input(..) function in Python gets a string from user.
        + EX: int(input(..)) gets an integer from the user
    - output: printing things to the screen
        - always prints characters, including the 'characters' 0-9, punctuation, newlines ("\n"), etc.
        - ***representing a string is different than printing a string!***

Writing Nice Code

- comments: anything following a hash # until the end of that line of text is ignored by Python.
    - you can explain in English what the nearby code is trying to do.
    - multiline comments: a triple-quoted string expression (not part of a statement) will be ignored by Python, like a comment. (Actually creates a docstring; prefer writing many #-comment lines.)
- good (descriptive) identifier names: helps self-document your code.
- breaking up long lines of code:
    - can't just put part of a statement on next line: Python will assume the lines are separate statements.
    - If one line of code is really long, consider using multiple assignment statements to get the job done.
    - If a line of code must contain enough stuff that it gets long, you can end the line with a backslash \ to indicate the statement continues on the following line as well.
        + EX: example of building up a long string.

Errors

- when we ask Python to do something that doesn't make sense somehow, we get an error.
- syntax errors: like spelling errors, missing operators (for example, *). Python couldn't even figure out what we were trying to say.
    + EX: 2(3+4) should have been 2*(3+4)
    + EX: (1+2) * (3+4)-5) has unbalanced parentheses. Should have been (1+2) * ((3+4)-5)
- run-time errors: Python understood what we asked it to do, but it turned out that it couldn't do it.
    + EX: dividing by zero
    + EX: int("eh, steve!") we expected numerical characters instead of "eh, steve!". Python couldn't get us a number.
- logic errors: The code we wrote is valid/legal, but doesn't do what we intended.
    + EX: inches = feet * 10 Even though this is valid code, we meant 12 instead of 10.

Strings

- Python value that is an ordered group of characters. Strings are sequences.
- Often our goal is to print them.

- we know about numbers and operations that combine numbers (+-*/**//); we can do the same with strings.
  - concatenate: +. This just adds two strings together to make a new, longer string.
  - repetition: *. string*int means to concatenate that many **copies** of the string together.
- Python will not allow us to treat the string `"15"` and the number `15` the same. `"15" != 15` .
  - we can convert between them: `int(s)`, `float(s)`, `str(n)`, etc. `int("15") == 15` .

- writing strings:
  - Python allows `'single'` `"double"` `or` `'''both kinds'''` `"""of triple quotes"""` for representing strings.
  - they all create the same strings (doesn't matter which one you use, same string value)
    + EX: 'asdf' == "asdf" → True
    + EX: 'mark\'s chinchilla' == "mark's chinchilla" → True

- special representations in strings:
  - newlines("\n"), tabs("\t"), quotes("\"" or '\''), and so on: special 'escape' sequences.
  - typed tabs are automatically converted to "\t" when Python stores these string values.
    + EX: chart of escape sequences.
    + EX: a triple-quote string can handle typed linebreaks. (They are also converted to "\n").

- string values versus printing string values
  - string value is a representation of the characters. Could include things like \n, \t, \' ...
  - printing a string means to actually place those characters on the screen. instead of seeing \t, we see 'enough' whitespace for a tab. instead of seeing \n, we see the rest of the string appear on the next line.
    + EX: >>>"hello" vs. >>>print("hello")
    (notice that the quotes are only used to define the string, they aren't actually part of the string; 1st we get back the string value with quotes shown b/c it's a string value; 2nd we see the characters printed).
    + EX: >>>"a\n\tb" vs >>> print("a\n\tb")
    + EX: >>>"Harga's House of Ribs" vs >>> print ("Harga's House of Ribs")

- we can assign string values to variables as well!
  + EX: `myName = "George Mason"`

- get the length of a string (or other sequences): len() function.
  + EX:
```
>>> s = "hello"
>>> len(s)
5
```

Algorithms and Flowcharts

- algorithm: approach to a task/problem
    - the idea or concept behind solving the task/problem
    - can be recorded different ways--written English, flow chart, code, pseudocode, pictures...
- flow chart: a drawing with boxes (actions), diamonds (branching questions), and arrows. Represents algorithm.
    - Branches translate to selection statements(if-elif-else). Cycles translate to loops (for, while).
    - suggestion: draw a flowchart first to solve a problem, then translate it to code
- what's the difference?

  → An algorithm is the idea, while flow charts are a means of writing out an algorithm.
    - so algorithms are often written/recorded by flow charts.

Booleans

- the bool type is just the set {True,False}.
- Boolean operators: **and**, **or**, **not**.
- Can write Boolean expressions with relational operators: < > <= >= == (which is not the same as the single =), != (the "not-equal-to" check)

  + EX: relational operator examples. Parentheses recommended.
- can use variables to store Booleans – very useful at times

Program Flow

- sequential: lines of code are executed in order, top-down. "waterfall" style.
    - This is the default, and is implicit (happens because multiple code lines at same indentation).
- selection statements: Boolean values are evaluated, and different blocks of code run based on whether the value was True or False.
    - we use if, elif, and else blocks to create the branches and conditions.
        + EX: show an if-else example's code, run it with different inputs to reach each branch.
- repetition: the same block of code is run over and over, either a set number of times, or until some condition is met.
    - Definite loop: use for loops to do task a set number of times (or for each element in a sequence).
    - Indefinite loop: use while loops to check a condition between each iteration, and as long as the condition is true each time, we execute the body of the loop again.
        + EX: show a while-loop example's code and run it.
            - "What's the Password?"
            - "enter an even number."
            - "keep dividing a number by 2 until the remainder is zero. ('How odd is this number?')"

Control Structures: CONDITIONALs

- let us write code that does different things when given different inputs.
- different, meaning "different lines of code" and not just like evaluating a mathematical expression with different inputs. That would be the same expression or code being executed.
  + EX: show flowchart with conditionals (diamonds).
- IF: contains a Boolean expression and an indented body of code.
  - syntax:
```
if booleanExpression :
    indentedStatements
    areTheBody
```
  - semantics:
    - evaluate the Boolean expression. if it's True, execute the body of indented code and then continue. If it's False, SKIP the body of code and continue.
      → decision whether or not to run a block of code.
- IF-ELSE:
  - syntax:
```
if booleanExpression:
  indentedCode
  runWhenBooleanExprWasTrue
else:
  secondCodeBlock
  whichIsRunWhenBooleanExprWasFalse
```
  - semantics: given a boolean expression and TWO blocks of code, if the Boolean expression is True execute the first block. If it is False, execute the second block.
    → decision which block of code to run.
  - notice that exactly one of the two blocks will run.

- IF-ELIF(-ELSE):
  - syntax:
```
if boolExpr1:
  block1
elif boolExpr2:
  block2
elif boolExpr3:
  block3
...
elif boolExprN:
  blockN
else:
  blockElse
```
    (the last two lines-- else: blockElse--are optional).
  - semantics: the block of code with the first True Boolean expression is run, and all the others are skipped (both False ones before and any blocks after this True Boolean's block).
    - exactly one else block is allowed at the end; if present, and ALL boolean expressions in if- and elif- blocks were False, this block will run. If there is no else block, then (just like a simple if-statement) when all boolean expressions are False, none of the code blocks will execute.
  - the else block is known as the 'default' block.
    + EX: Branching Summary table 3.2.

Sequences: Brief Intro (to be often used with for-loops)

- sequences are anything Python knows how to think of as an ordered set of values.
  - we already know strings are sequences of characters.
  - we can generate lists of numbers with the range(..) function. range(start,stop,step). First value (leftmost in resulting list) will be the start value; we add more values by adding step each time. As soon as we reach or surpass the stop value, we are done. The stop value is not included in the resulting list.
    + EX: `range(10) == range(0,10,1)` → `[0,1,2,3,4,5,6,7,8,9]`. Note: has10 elements, 0-9
    + EX: `range(3,7)== range(3, 7,1)` → `[3,4,5,6]`. Again, note stop value (7) isn't in the list!
    + EX: `range(0, 50, 10)` → `[0,10,20,30,40]`. Step (10) is the increment value.

Control Structures: REPETITIONs

- let us write a block of code that will be repeated multiple times, based on some condition (a Boolean expression that is evaluated each time).
- some repetition structures are executed a specific number of times (definite loops).
- some repetition structure are executed indefinitely, as many times as the condition is True (indefinite loops).
- "iteration": one single execution of the loop's body. A loop usually performs multiple iterations.
- WHILE:
  - syntax:
```
while boolExpr:
    bodyOfWhileLoop
```
  - semantics: given a Boolean expression and a body, Python checks the Boolean expression. If it was True, Python executes the body and then comes back to check the Boolean expression again. Each time the condition is True, we run the body again. The first time that the Boolean expression is False, Python is done executing the while-loop, skips to after the entire while-structure, and continues with the code after it.
    - it's possible that the body runs zero times, if the Boolean expression is False the first time.
    - so we'd better be sure that the body of the loop actually has code that could make the condition False eventually! (If not, the loop is 'infinite').
      → We want our "sentinel" variable to be modified.
    - Interactive mode: CTRL-C tells Python to stop executing the code, and return control to you with a new >>> prompt.
    - file interpretation mode: CTRL-C forces Python to quit when we get stuck in a loop.
      - in Windows cmd: you might need to use CTRL-Z.

- FOR-Loop:
  - ***syntax***:
```
for varName in someSequence:
    bodyOfForStmt
```
  - ***semantics***: given a sequence of values and a name to use for each value, run the body of code where that variable name holds one specific value from the sequence. Do this for each item in the sequence.
  - to run the body of code a set number of times, use range(#) to say how many times to evaluate the body.
    +EX:
```
for i in range(10):
    print(i)
```
```
sum=0
for val in [10,5,3,2]:
    sum += val
```
  - we can use our loop variable (called i, in the above code) to know which time is currently running
    + EX: print something 10 times
    + EX: calculate something by inspecting each value in the sequence, like sum in the 2nd example.
    + EX: print the numbers 1-1000 NOT on separate lines (use an 'accumulator' string variable)

- special control flow in repetition blocks
  - continue statement: tells Python to stop going through the current loop iteration, and start the next iteration (checking the condition on a while loop, getting the next item in the sequence for a for-loop).
  - break statement: tells Python to immediately stop going through the current iteration, as well as to just exit the whole loop once and for all. (doesn't check while loop's conditional, doesn't go back for more elements in for-loop).
  - both continue and break are associated with the closest-nested while or for loop.
  - they almost always show up inside some conditional block (if/elif/else).
    - It's kindof pointless to 'always' have it in a loop, because otherwise we'd never reach code after it in the loop (continue), and we'd never reach a second iteration (break).

Sequence Operations (i.e., things you can do to lists, strings, and other (ordered) sequences of values.)

- **indexing**:
  - zero-based indexing: from left to right, spots are numbered 0,1,2,3,4…
  - negative indexing: from right to left, spots are also numbered -1, -2, -3, -4, …
  - +EX:
    ```
    >>> tup = (5,2,True,"hello world", 7,9)
    >>> tup[-1]
    9
    >>> tup[2]
    True
    >>> tup[-3]
    "hello world"
    ```
  - index position must exist, or else it's an error.

- **slicing**: xs[starting_index : stopping_index : step_amount]
  - allows us to grab a sub-sequence. We give start:stop:step values, much like the range function's arguments.
  - grabs a 'slice' of the sequence, resulting in a new sequence *of the same type* (tuple, string, list, etc).
    - even if the length of a slice is one or zero, the type is preserved
  - include starting index, increment by step_amount, until you reach/pass the stopping_index. (Don't include stopping_index in result, just like the range function).
    - default step_amount is 1, default starting_index is 0. (just like range())
    - EX:
      ```
      xs = [2,4,6,8,10,12,14,16,18,20]
      xs[1:6:2] == [4,8,12]
      ```
    - step_amount can be negative
      - Only way for slice to go right-to-left. (neg. starts/stops alone aren't sufficient)
  - if start/stop/step request more than tuple contains, we just get back the portion that did exist -- it's not an error.
  - you can indicate "from beginning" and "to end" by omitting either start or stop index:
    - xs[:5] ==> from beginning up to but not including index 5. Same as xs[0:5]
    - xs[2:] ==> from index 2 to very end of sequence Same as xs[2:len(xs)]
    - xs[:] ==> from beginning to end Same as xs[0:len(xs)]
    - xs[::-1] ==> generates a reversed copy of the list! same as xs[len(xs):-(len(xs)+1):-1]
      - slice syntax as lefthand side of assignment:
        - replace those spots in slice with given new sequence (without changing id).
        - only possible for mutable values (lists)
        - this is an *update* of the sequence, not a re-assignment.
      - entire slice on righthand side of assignment:
        - generate a *copy* of the entire sequence.
      - you can perform slices/indexes one after another to 'reach' deeper into some data structure:

```
xs = [0,5,12,[0,1,2,3,4,(2,5,"hello there", True),6,7], 45, 4, 2]
# notice the stop value isn't included:
xs[3][3:6] == [3,4,(2,5,"hello there", True)]
xs[3][5][2] == "hello there"
xs[3][3:6][2][2] == "hello there"
# slices can occur one-after-another! This one's kind of ridiculous.
xs[3][:][3:6][1:][1][:][2][:] == "hello there"
# h is at index 7 in the string, so don't include following e.
xs[3][3:6][2][2][1:8:2] == "el h"
```

- "set membership": val in seq
- "set non-membership": val not in seq

- + (concatenation)
  - note: creates a (shallow) **copy** from the original sequences.
- * (multiplication). Concatenate that many copies. Just like mathematical multiplication: x*3 = x+x+x.
  - 4 * ('a','b','c') == ('a','b','c','a','b','c','a','b','c','a','b','c')
- len: how many elements in the tuple? Not the same as 'what is the last valid index?' (len is one greater in answer than "what's the last valid index?").
  - EX: len ([0,2,4,6,8]) == 5
- min / max (only use with all elements the same type, such as all numbers)

Lists

- complex data type (has sub-components; can be any python value)
- sequence (sub-values are ordered; can index/slice)
- mutable (CAN be modified after creation)
- declaration: square brackets [ ] with commas between values.
  - empty list: [ ]
  - length-one list: [val]
  - many vals: [a,b,c,d,e,etc]
- Note: [ ] isn't equivalent to [[ ]]. (similar to how 3 != "3", 3!= [3], etc.)
  - [] is the empty list. [[]] is the list with one element, where the one element is the empty list.
- LOOPS

  - you can use a list directly in a for-loop, getting the list's *values* in order for your loop-variable:

    + EX:
```
xs = [1,1,2,3,5,8,13]
fibonacci_sum = 0
for val in xs:
    fibonacci_sum += val # or some other val usage
```

- more flexibility: use `range()` function to grab the *indexes* in your preferred order:

    + EX:
```
for i in range(len(xs)):
    print ( xs[i] )
```

- Mutable Sequence Operations: lists are mutable, so we can perform these operations on lists.
  - index assignment: replaces the single element at indicated index with value that is assigned.
    - EX : xs[3] = x + 32
  - slice assignment: replaces indicated slice with the list value that is assigned.
    - EX: xs[1:10:2] = [2,4,6,8,5]
    - can 'slice' everything: xs[:].
    - replacement slice doesn't have to have the same length, bigger or smaller also work
      - (except when step!=1, then lengths must match).
    - del: removes the specified spots from the list.
      - del xs[i] removes position i
      - del[i:j] or del[i:j:k] removes all positions that slice describes
    - methods: append, extend, count, index, insert, pop, remove, reverse, sort
      → be comfortable with append/extend, index, pop, sort.

- For-loops in multiple dimensions:
  - given multiple dimensions, you can loop through each in turn.
    - you still choose whether to get the values themselves, or use indexes manually (more control)
      - EX:

```
xs = [ [0,1,2,3,4], [5,6,7,8,9], [10,11,12,13,14], [15,16,17,18,19] ]
for fivenums in xs:
      for num in fivenums:
            print (num)
```

output ==> 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 *(on separate lines)*

```
for i in range(len(xs)):
      for j in range(len(xs[i])):
            print (xs[i][j])
```

      - we have to index into xs as xs[i] to get the length of the sub-list via len(xs[i]).
      - we have to index down to xs[i][j], but we can now get to both the value and have the indexes.

Tuples

- complex data type (has sub-components which can be any python value)
- sequence (sub-values are ordered; we can index/slice)
- immutable (cannot be modified after creation)
- declaration: use commas (parentheses alone are not enough!)
  + EX: one-tuple: (val,) or val,
  + EX: triple: (a,b,c) or a,b,c

- LOOPS and tuples
  - you can use a tuple directly in a for-loop, getting the tuple's values for your loop-variable:
    + EX:

```
tups = [(1,2,3),(4,5,6),(7,8,9)]
for (a,b,c) in tups:
      print (a+b/c) # or some other usage of a,b,c
```

  - more flexibility: use range() function to grab the indexes in your preferred order
    + EX:

```
for i in range(len(tup)-1, -1, -1):      # access in reverse order
      print (tup[i])
```

Usage: Tuples vs. Lists

- lists:
  - generally homogeneous (all values in it are the same type)
  - used for mutability, tend to update or modify frequently.
- tuples:
  - immutable, so length also can't change.
  - tend to be heterogeneous (values in it are not the same type)
  - good for grouping values together: returning values, getting many values from user, etc.

Mutability, References

- mutability means that the value can *change*.
- strings, tuples, and others are immutable, and couldn't change – each 'modification' created a new value by copying from the old value as necessary.
- now that we see list values can change without making new copies, we need to tell copies/aliases apart.
    - each Python value gets its own unique address, which the built-in id() function reports.
    - mutability's about the contents of a memory location, not a variable that happens to refer to that memory location.
        + EX:

```
>>> xs = [1,2,3]
>>> ys = xs
>>> (id(xs),  id (ys)  )          # look at both id's at once
(8048234, 8048234)                # you'll see diff. nums, but they'll match
```

    - re-assigning versus updating a variable:
        - updating a variable replaces one part of the variable with a new value, leaving the variable's id-number unchanged.
        - works for updating an entire slice as well--no change in id.
            + EX:

```
xs[5] = 75
xs[2:8] = [7,6,5,6,7]
```

                - notice the replacement slice's length doesn't have to match what it replaces (unless the step isn't 1).
        - re-assigning a variable does change the id# for a variable – it now "points" to the new value.
            + EX:

```
xs = [1,2,4,3,5]          # the list [1,2,3,4,5] is new: assigned to xs.
xs = xs[:]
ys = xs
```

        - Note: xs[:] creates an entire slice, so it is a copy. So xs is re-assigned to the copy (id changes).
        - ys is re-assigned the value of xs, so whatever value ys happened to point to before is forgotten, and it now points to whatever xs did. The old ys value now has no association with ys, and xs and ys now have the same id-number. (because the value they both point to has that id-number).
        - shallow copies: given a complex value of complex values, when a complex value (e.g. sequence) is 'copied', only the reference to it is copied. Thus any references that get copied will become duplicates:
            + EX:

```
onex = [5]
xs = [onex] * 3
print (xs)      # prints [[5],[5],[5]]
xs[1][0] = 3
print (xs)      # prints [[3],[3],[3]]; all refs saw the change.
```

            - whatever xs[1][0] points to gets changed to [3]. It turns out that's what xs[0][0], xs[2][0], and onex[0] also point to, so they all witness the change.

Test 2 Review Guide CS 112

Bring your GMU ID and a pencil to the test.

This test will be focused on the materials covered after the contents of our first test. We can't entirely avoid the materials from the first test, as loops and variables and whatnot are of course necessary components to showcase the new materials. The format will be the same as our previous test.

Take a look at any Practice Problems, TopHat questions, all code samples posted on Piazza, and of course all lab materials, for some more directed review questions on the topics that were hardest for you. ***Open up an interpreter and try out all these concepts***. The class focus is still on being able to do programming.

This is perhaps more detailed than you want—-but that just means you've got the extra details when you want them. We don't intend to test every sub-bullet on this review through the test, so be sure you keep focusing on writing/testing code to learn about these broad topics and feel comfortable discussing these topics.

Good Luck Studying!

A Bit More Details Than Last Time....

- We will need to test the use of **lists and loops** together a bit more than we already did last time; please look to our previous review guide for details on looping over the items in a sequence, index-loops vs value loops, looping over structures in multiple dimensions, and calculating properties over lists of values.
- we also need to cover a bit more with **mutability** of lists. Look for practice by creating lists, aliases to lists, modifying cells, drawing out the memory, running the visualizer, and seeing what does or doesn't change.

Formatting Strings

- Percent operator: given a string with placeholders (such as %d, %f, %f), and a tuple of replacement values, Python substitutes through to generate a new string. See **LIB 4.7.1** for more info.
- + EX:
  - "we are the %d %s." % (3, "musketeers") ➔ "we are the 3 musketeers."
  - "your change is $%.2f, thanks." % (6/7) ➔ "your change is $0.86, thanks."
  - "%d, %f, %s" % (3.7, 3.7, 3.7) ➔ "3, 3.700000, 3.7"

  - Know these:
    - %d (for decimal numbers → ints).
    - %f (basically for our floats).
    - %g (prefers to write in scientific notation).
    - %s (strings).
    - %% represents a % in the output, much like \\ represents a \ in string literals.

    - extra modifications: %m.nf: m is the minimum number of columns to be used in the substitution; extra spaces are used to pad up to this amount if needed. n is the number of columns we must see after the decimal point. Rounding is performed in an effort to faithfully represent the quantity.

- Format method: much like the percent operator, but uses {}'s for each placeholder, and arguments instead of a tuple operand.
    - see **LIB 6.1.3.2** for more details. We're brief here, but it's as important as the % operator.

Functions

- **Benefits** of having functions:
    - Give us a way to abstract: hide implementation details behind a name. A sort of factory that accepts some inputs (arguments) and generates an output (return value), perhaps causing a few mutations of data along the way.
    - Decomposing a complicated task into sub-tasks: improved readability, easier to test/debug/upgrade.
    - Code reuse: define the function once and call it as often as we need.
    - EX: the int() function can be given a string; it picks through our string, finding the number represented inside the string, and returns that value. All we have to know is that it turns a str into an int for us.

- **Defining** a function: allows a programmer to specify what code to run when requested, defining the expected behavior and result.
    - Syntax:

```
def theFuncName (formal,parameters,list,here,if,any):
    functionBody ( any statements, indented)
    maybe a return statement somewhere
```

    + EX:

```
# function definition
def add2nums (x,y):
    z = x+y
  return z
# function call, inside a call to print
print(add2nums(5,6))                # prints 11
```

- **Calling** functions: already seen. Function's identifier, followed by arguments (expressions) inside parentheses.
    - Examples:
        - print("word",num,"hi!")
        - max3(a-1,b*5,c+2)
    - Allows programmer to perform the functionality associated with the task
    - Give function name, any required inputs (the 'argument list'), get back the result.
      → (Sometimes there's no result expected. Then, it returns the None value).
    - Operation of a function call:
        - Python locates the code for that function, and uses the arguments (assigns them to the parameters)
        - Now that the parameters have values, Python executes the body of the function, just like regular code.
        - Once Python reaches a return statement, the function call is done, and the return value is sent back to the original function call.
        - Unconditional branching: execution flow always changes due to function calls.
    - You can think of the original function call as having 'reduced' to the returned value of the function.

+ EX: print (add2nums(3,4) + 5)
---> Python takes the arguments (3,4), assigns x=3 and y=4, runs the body of add2nums (x+y is evaluated as 3+4, so 7 is stored in a local variable named z)
---> Python returns from the function call, and you can pretend the expression is 'reduced' to : print (7+5).
---> Python then continues by calculating 7+5 is 12, calling print with argument 12, and ends up printing 12. The function call and print statement are all done
---> Python would then continue with the next line of code

- **Return statement**: a return statement causes execution to immediately exit the function with a value. There are two forms:

```
return someExpr      # calculates the value of someExpr, returns it.
return               # returns the None value, since no other value was specified
```

- If the end of the body of the function is reached w/o a return stmt, the None value is implicitly returned.
- meaning: "Python, find the value of this expression; then, since we're executing a function, I'm telling you that you are done right now, and please return to where the function was called, and the result of that function call is the value you just calculated. Thanks!"

- **Parameters / arguments**
  - Parameters are the list of identifiers specified in function definitions: names of incoming values
  - Arguments are the list of values (expressions) provided in function calls.
  - Arguments will be evaluated to values first and only values are passed to parameters.
  - Different types:
  - **Positional parameters and arguments**. Parameters get their values based on the arguments' ordering in the function call. Arguments are required for positional parameters.
    - def some_function(first, second, third):
      return (first+second+third)
      x = some_function(10, 25, 50) # 10 is the 1st positional arg., assigned to the 1st positional param.
      # same for 25-for-second, 50-for-third.

  - **Default parameters**: default values provided in function definition. Arguments are optional. If no argument provided, use the default value.

    - def func(a, b, c, x=0, y=10, z="hello"):
      …
    - func(1,2,3) # three positionals given; then uses all three defaults (for x, y, z).
    - func(1,2,3,4) # four positionals used for a,b,c,x. defaults for y, z used.
    - func(1,2,3,5) # four positionals used for a,b,c,x. (y,z defaults). Can't skip over early
      # defaults with positionality alone! (Python can't read your mind which
      # one to use)
    - func(1,2,3,4,5,"six")# six positional arguments, used for a,b,c,x,y,z. No defaults used.

  - **Keyword argument**: specify a parameter name with the provided value at function call. Values passed to parameters using the explicit name regardless of the order.
    - def func(a,b=3,c=5): …
    - func(2,c=20, b=15) # one positional arg (used for a); the other two given as keyword arguments.
    - func3(a=12, b=13, c=14) # all three given as keyword arguments.
    - func3(c=6, b=5, a=4) # all three given as keyword arguments, in arbitrarily chosen order.
    - func3(14, c=16) # positional 14 used for a; keyword c=16; default 3 used for b.

- **Scope**

  - **Global variable:** defined outside any function in a file. Visible starting from the definition point to the end of the file. Visible inside functions unless hidden by a local variable with the same name.
    - **global** statement can be used to explicitly require access to a global variable or even create one.
    - Try to restrict the uses to global constants.
    - Recipe of removing globals
  - **Local variable:** defined inside a function. Only visible inside the function, "dies" when we exit the function.
  - **Scope resolution:** the process of searching for a name in the available namespaces.

Mutability

- Property of complex/container values: they can be updated. Lists, sets, and dictionaries are examples.
- Main concern: finding aliases between two variables from caller's scope and called function's scope, realizing when a value found by one variable name is modified by the alias (other name).
- Mutability works across function calls.

  + EX: mutafunc.py

```
| def change_me(xs):
|    xs[2] = "hi"
| def main():
|   my_list = [1,2,3,4]
|   change_me(my_list)          # update @ pos. 2 mutates my_list
|   print (my_list)
|   my_list = [1,2,3,4]         # restore contents
|   change_me(my_list[:])       # slice [:] creates a copy, so only the copy is modified!
|   print (my_list)
| main()
| ... output at the terminal:
| [1,2,"hi",4]
| [1,2,3,4]
```

Modules

- A module represents a grouping of Python definitions.
- Pragmatically, it's one Python source file's contents.
- Using a module's contents: import them.

  - import all with a single name prefix (as 'qualified' names):
    + EX: **import module_name**
    - Python finds a file named module_name.py, and now in the rest of the current file, module_name.whatever will allow us to refer to (use) something named whatever inside the module_name file.
    - called 'qualified' because all names must first be 'qualified' with "module_name ."

  - import all definitions in a module directly to our namespace (as 'unqualified' names):
    + EX: **from module_name import ***
    - Python again finds a file named module_name.py, and now in the rest of the current file, the name whatever can be used to allow us to refer to module_name.py's whatever definition.
    - called 'unqualified' because there are no extra steps ('qualifications') to get the name: just type whatever directly.
      - note that we can't use module_name.whatever; just whatever.
      - This can be convenient because there's less typing
      - This can be inconvenient because now we can't (re)use any names that happened to be used in that other file without confusion about which one is intended. May lead to bugs!

  - import just a few things from another file, as unqualified names:
    + EX: **from math import (sqrt,pi,e)**
    - only these three things are imported from math.py. They are unqualified names.

Sets/Frozensets

- We looked at the documentation for Python sets to learn how to read documentation.
- Implements the mathematical notion of sets: a grouping of values, no implied order, no duplicates allowed.
- Creation: comma-separated values in braces, like {1,2,3} or {"hi","there","hi"}.
  → but {} always means empty dictionary; use set() to get the empty set.
- Some available operations: (given a set named s)
  - len(s)
  - val in s , val not in s
  - min(s) , max(s)

- Some available methods <====> *and the convenient operator version:*
  - isdisjoint(other)
  - issubset(other) <====> set <= other
  - issuperset(other) <====> set >= other
  - union(other,...) <====> s1 | s2 | …
  - intersection(other,...) <====> s1 & s2 & …
  - difference(other,...) <====> s1 - s2 - …
  - symmetric_difference(other) <====> s1 ^ s2
  - copy() (gives a shallow copy of the set or frozenset)

- Some available methods for sets only (not frozensets):
  - update(other,...) <====> s1 |= s2 | s3 | …
  - intersection_update(other,...) <====> s1 &= s2 & s3 & …
  - difference_update(other,...) <====> s1 -= s2 - s3 - …
  - symmetric difference_update(other)<====> s1 ^= s2
  - add(elem) # adds elem to set (no effect if already present)
  - remove(elem) # removes elem from set. KeyError when not found.
  - discard(elem) # like remove, but fails silently
  - pop() # removes/returns arbitrary elem from set.
  - clear() # removes all elements from set.

- Useful trick: if you want to get rid of duplicates from a list, convert it to a set and back to a list:

```
xs = [1,1,5,3,4,2,3,6,7]
xs[:] = list(set(xs))
print(xs)      # prints [1,2,3,4,5,6,7].  Note, sorting is not guaranteed! original order is lost.
```

Dictionaries

- Represents key-value pairs.
  - keys can be any 'hashable' (essentially, immutable all the way down) value. E.g., strings, Booleans, numbers, tuples that only contain hashable values.
  - values can be any Python value – mutable or not, function definitions, objects, anything!
- Container/complex data type (contains sub-values)
- Mapping data type (no ordering; just "maps" inputs to outputs)
- Mutable (contents may change by adding/removing key-value pairs, or by updating a key-value pair's value.)

- Declaration: in curly braces, key-value pairs associated with a colon, and separated by commas.
  - direct: d = { key1:val1, key2:val2, key3:val3 }
  - empty: d = {}
  - via a list of length-two sequences: d = dict ([[k1,v1], [k2,v2], [k3,v3] ...])
  - via keyword arguments (only unquoted-strings as keys): d = dict(k1=val1, k2=val2, k3=val3…)
    - note that dict is a built-in function; please don't use dict as a variable name.
  - +EX:
```
d = {"hi":(1,2,3),"bye":4.5}
d = dict ( [["hi",(1,2,3)],["bye", 4.5]])
d = dict (hi=(1,2,3),bye=4.5)  # Note these three create equivalent dictionaries
```

- Access: looks like indexing into a sequence, but any valid key may be used.
  + EX: d["hi"] d[4] d[(i,j)]
    - get method: akin to indexing, but returns default value when the key isn't found rather than crashing. The default value is None.
      + EX: d.get(word,0)

- Use in for-loops:
  - the keys are assigned to the loop-variable. *(order of keys is arbitrary!)*
    + EX:
```
d = {"a":3, "b":1, "c":2}
for k in d:
        # you can access the key as k, and the value as d[k]
        print (k, "=>", d[k])
```
    - prints out:
```
a => 3
c => 2          # notice the unusual order! Dictionaries are not in some sorted order.
b => 1
```

- Useful methods:
  - keys(): returns sequence of keys. (won't be in any particular order!)
  - values(): returns sequence of values. (Guaranteed same order as keys()).
  - items(): returns sequence of length-two tuples: they are (key,value) pairs. (also same order as from keys()).
  - pop(k): removes key-value pair where k is the key. returns the value.
    - error when key not found. (Or, give second argument as default return value)
  - popitem(): arbitrarily chooses a key-value pair; removes it from the dictionary, returns as (key,val) tuple.

File Input/Output (File I/O)

- Python can read from, and write to, files that contain just text (sequentially-stored ASCII codes). Approach:
  (1) create reference to file, also choosing mode (reading, writing, append…)
  (2) perform all your reads/writes
  (3) close the file
- File reference: Python handles all the tricky low-level details, and provides a public interface to us by letting us get a File object, with methods available to read/write strings.
  - We only have to know how to use this public interface (by calling the methods)
    - good abstraction and encapsulation!
      - we don't know how the details are implemented inside, so it's an abstraction
      - we don't have to deal with files any other way. this interface is well-encapsulated.
- Mode: denotes how we may use the file 'stream'.
  - 'r': indicates reading may occur. (IOError when we try to read a non-existent file).
  - 'w': indicates writing may occur. (overwrites file if it exists; creates new file otherwise).
  - 'a': indicates appending may occur. (like writing, but adds to end of file without overwriting).
  - **(extra details not covered in class or on this test:)**
    - "+": indicates mixed mode. (so, writing/reading/reading also allowed for "r"/"w"/"a"
      - behavior (error/overwriting/etc) still indicated by main mode.
        + EX: "r+" requires file to exist.
    - U: when added to 'r' mode, indicates that all versions of newlines should be represented as "\n".
      - default behavior in Python 3. (Mode deprecated since 3.4)
      - great to just always use: simplified explanation: various system representations for a newline, but we don't want our code to have to tell the difference. Python only presents \n to us in this case.
- Reading: methods (usable on file references only; need "r" in mode)
  → all reading methods keep track of what next character would be. Successive read() or readline() calls will get different contents each time! Return empty string if EOF (end-of-file) is reached.
  - read(): read/return entire file's contents, return it as one string. (will include newlines as found in the file)
  - read(x): read/return x bytes/characters from file
  - readline(): read/return characters found until next newline character (includes the '\n')
  - readline(x): read/return up to x bytes/characters, stopping early if/when newline character found
  - readlines(): reads whole file, splits by newlines. Return list of these 'lines' of the file,including newline chars.
- Writing: methods (usable on file references only; need "w" or "a" in mode)→ *never implicitly adds newlines, ever!*
  - write(s): writes string s at current file position.
    - doesn't matter if s contains newlines or not, they are just characters that can be written.
    - it's safe to write any small bit to the file at a time with its own write() call.
  - writelines(xs): xs must be a list of strings. Each of these strings is written to the file in order.
    - convenient when you already have constructed all the lines and just want to write them all.
    - writelines requires the desired newline characters to be present in your lines.
      → This is just like writing a for-loop that uses write() on each element of your xs.
- tell(). *(we didn't do much with tell)*
  - get an integer as the 'absolute' index into the file for the current file position.
  - all reads/writes are based on this location, so it's useful to figure out where we are.
- seek(i, whence). *(we didn't do much with seek)*
  - i is the new index we want the reads/writes to occur from next.
  - whence is the style of index:
    - whence=0: means absolute indexing (from beginning of file). Changes current position to be the index.
    - whence=1: means relative indexing (add index to current position for new file position).
    - whence=2: means end-relative indexing. Need a negative number, but can index back from end.

Final Exam Review Guide CS 112

Bring your GMU ID and a pencil to the test.

The final exam in this course will be cumulative, with an emphasis on the newest materials (slightly more than 1/3 of the final). So be sure to also use the review guide from the previous two tests to brush up on the older materials. The older materials should feel easy by now, though, so those questions should help boost your grade. Take a look at any Practice Problems, TopHat questions, all code samples posted on Piazza, and of course all lab materials, for some more directed review questions on the topics that were hardest for you. Open up an interpreter and try out all these concepts. The class focus is still on being able to do programming – the final exam will be formatted the same as our previous tests. No surprises, just some new materials.

This is perhaps more detailed than you want—-but that just means you've got the extra details when you want them. We don't intend to test every sub-bullet on this review through the final, so be sure you keep focusing on writing/testing code to learn about these broad topics and feel comfortable discussing these topics.

Good Luck Studying!

Polymorphic parameters

not on exam

- single star: collapsing or expanding positional arguments into a sequence as a single parameter
  - in function definition's parameter list: a single star means to accept any number of positional arguments, and put them in order in a tuple with the parameter name
  - in a function call: a single star means to 'explode' the sequence into positional arguments. Called function acts just as if it received all values individually, so the exact number of positional arguments needs to be acceptable
- double star: expanding a dictionary or collapsing multiple keyword arguments into a dictionary
  - in function definition's parameter list: a double star means to accept any number of keyword arguments, and put them in a dictionary with the parameter name. Must follow all positionals, defaults, and single-star parameters
  - in a function call (preceding dictionary expression): a double star means to 'explode' the dictionary into keyword arguments. Called function must be ready to accept these particular keyword arguments' names. Extra arguments cause errors.

Basic Object-Oriented Concepts

- Object:
  - represents some entity/thing in the world/system you are implementing
  - has associated state (called attributes or instance variables), and associated behaviors (methods)

- Encapsulation/Information Hiding:
  - by grouping data (attributes) and behaviors (methods) together, we define a logical boundary.
  - each object 'encapsulates' its data/behaviors. Related data/code is in one place.

- Abstraction:
  - provides a simplified view of things: details of implementation are contained inside, and the user doesn't have to bother with those details.
  - the user only sees the "public interface", meaning the public attributes and methods of the class.
  - encapsulation partly implements abstraction by hiding those details within the logical boundary of the object. Further making things private provides even better abstraction.

Python Classes

- Allow Python programmers to add their own types to the language.
  - lets us group values (as instance variables), creating a complex data type. From this perspective, a class defines a new type that is just a "fancy container" with named sub-values.
  - lets us group functionality (via method definitions), only allowing these methods to be called on our type.
  - quite powerful way to introduce abstractions and use encapsulation

- **Three basic phases:**
  - Class definition:
    - one-time "blueprint" definition that says what values (objects) of this type will contain
      - what instance variables? what methods?
      - the class name is our new Python type. (E.g., Circle, Person, Button, etc)
  - Object instantiation:
    - can be done multiple times. This uses the class definition to create a Python value that truly is a value of our newly defined type.
    - each object instantiation creates a separate, distinct value, at its own unique spot in memory.
      - guaranteed to have all attributes/methods from class definition
        - but it has its own particular values for the attributes
      - to create an object, we must call the constructor (the __init__ method).
        - looks like a function call, using the class name as the function's identifier name.
          → Examples: ValueError("☹") Triangle(3,4,5) Person("George",20)
        - But really it's calling the __init__ method of that class definition.
  - Object manipulation (usage):
    - Accessing/changing attributes of an object (e.g., my_point . x = 0)
    - Calling methods of an object (which likely modify the attributes of that object).
- Attributes: (also called 'instance variables'):
  - when defining what objects of a class are, we need to define what values/state each distinct object will keep track of. We call these values 'attributes'. Each object is called an instance of the class, so each attribute is called an instance variable.
  - to create an attribute, assign it as a dot-qualified identifier of the self object in the __init__ constructor:
    + EX: self.my_attribute_name = some_value
    - you should only create attributes in your __init__ method. Python allows it other places, but this leads to bad coding styles in all but a few circumstances. Please pretend it only works in the __init__ method

- to access an attribute, you need an object of that class, and use the dot-operator to access it:
  + EX:

```
c1 = Circle(5)              # first, create a Circle object named c1.
print(c1.radius)        # request radius attribute's value of this specific object.
                                    # → we see that its value was 5.
```

- private attributes:
  - Python allows for hidden attributes. Choose a name that begins with __, such as __radius or __account_balance.
  - the usual dot-operator way of looking up the attribute no longer works anywhere outside the class definition – writing .__attributeName would only work inside the class definition.
  - provides abstraction, in that the user of the class can't access/know about this value.
  - to selectively let the user read or write to a private variable, just provide a method (such as get_radius or set_radius) that can carefully make other calculations in deciding how to let them see or modify the private attribute (if at all). Being in the class definition, these methods are allowed to use/view/modify the private variable with its __actual_name.

- complex attributes:
  - there is no restriction on what sorts of values may be attributes. Other objects, lists, tuples, anything can be an attribute. You can 'stack' them as deeply as you want. (We call this technique of objects having attributes that refer to other objects as 'aggregation').

Methods:

- defined to describe behavior of objects
- ALL methods must have at least one required parameter, traditionally named as 'self'
  - for a constructor, Python creates the object for us to use as self.
  - for ALL other methods, the self parameter is what is used to call the method.
- all instances of a class share the same group of methods, but each time a method will only be invoked for a particular instance
  - usually a method is called with the dot-operator: obj.method(arg_list)
  - implicitly passing an alias of the object the method is called for (obj) to the special parameter 'self'
  - will be able to access/update the instance variables of that particular object inside method via 'self'
- additional incoming values/return value work the same way as before
  - a constructor always returns a reference to the newly created object
- special methods: __init__(), __str__()
  - the __str__ method lets Python get a user-friendly string version of an object.
    - Built-in function str() looks for a definition of __str__() and executes it
    - note that it returns a string, it performs no printing! assemble + return string
- Creating a class:
  + EX:

```
class Circle(object):
        def __init__(self, rad):          one instance radius
                self.radius = rad
        def __str__(self):
                return "Circle: radius=%d" % self.radius
        def get_diameter(self):
                return (self.radius * 2)
```

* how to create instances

- Notes from Example:
  - the __init__ method is what lets us later create Circle objects.
    + EX: c1 = Circle(5)
  - by calling a 'function' named Circle(), we are actually calling the Circle class's constructor method __init__()
  - Argument 5 matches up with the 'rad' parameter.

        - it actually would have been more common to have named rad as radius, and then create the attribute thus. *(naming params the same as the targeted instance variables is convention)*
            + EX: `self.radius = radius`
    - we would say that c1 is an "instance" of the Circle class, so c1 is an object.
    - we could create many, many Circle objects by calling the Circle constructor again and again. There would be many objects, at different memory locations, all instances of the Circle class. But there is always just one class definition.
    - we can create as many methods as we want (beyond __init__ and __str__).
    - Look at __init__'s and get_diameter's original definitions above to see how 'self' is used.
        - + EX:
```
c1 = Circle ( 5 )
d = c1.get_diameter()
```
            - we ask the object c1 to call get_diameter(), and so c1 is the argument that represents self.

Functions versus Methods

- Functions are stand-alone, in the sense that its behavior is entirely defined by its parameters.
    - functions can be thought of as the mathematical notion of a (partial) function, being strictly a mapping from inputs to outputs. This notion is violated through usage of globals--one of the reasons we avoid them.

- Methods tend to require some state/data to operate, like the sort method needing the list to be sorted.
    - methods can *only* be called on objects of the class where the method was defined: e.g., the get_diameter method from our Circle example can only be called upon a Circle object (such as c1.get_diameter()).

- Technically you could always write a function instead of a method, and just have the first parameter to the function be the object upon which you want to operate, but methods (and the change in syntax) give us a more natural way of indicating what is supposed to be happening; it lets us write all definitions related to a class in one place, and then requires us to only use the methods with objects of the class.

Exceptions describe + respond abnormal behaviors

- An exception is an action outside the normal control flow, representing some error or unusual event
    - not just if-else, for/while loops, or function/method calls
    - some built-in exceptions, and an example of what might cause it:
        - ***FileNotFoundError***: trying to read a non-existent file
        - ***IndexError***: attempting to use an invalid index (out of range)
        - ***KeyError***: attempting to use an invalid key with a dictionary
        - ***NameError***: a name (identifier) was used without being defined (or isn't accessible in this scope)
        - ***SyntaxError***: the string of characters in the source code file doesn't obey the Python language itself
        - ***TypeError***: Python needed a value of one type, but got another. doesn't have right type ex: int([1,2])
        - ***ValueError***: although Python was given the correct type of value, the particular value given wasn't acceptable (such as "abc" given to the float() function; it can handle strings, but not this one in particular) right type, value doesn't work
        - ***ZeroDivisionError***: dividing by zero had no valid answer to return ex: int("4.5")

- Exceptions: Step-by-step description
  1. Exceptional condition occurs (such as a division by zero)
  2. System raises the exception
     - exceptional control flow causes blocks of code to be immediately (abruptly) exited, instead of the usual next-line-of-code execution.
  3. Programmer handles the exception (or else the program crashes)
     - can ignore it ("defer"), run 'corrective' code ("handle" it with an except block), or log & ignore ("log and defer" by excepting it and re-raising it).

- try-except:
  - allows us to try code that might cause errors, w/o fatally crashing the program.
  - the except block(s) let(s) us recover from an error by running some code and resuming normal operations.
  - put the suspicious code in the try-block, and then add an except-block that handles the exception if it occurs.

  - Styles of try-except:
    - very general: catches all exceptions.

+ EX:
```
try:
        y=7/0
except:
        print ("bad division!")
```
+ EX:
```
try:
        y=7/0
except Exception as e:
        print ("bad stuff happened: ", e)
```

only if matching exception triggered

    - specific: catch only specific types of exceptions (only catches ZeroDivisionError exceptions)

+ EX:
```
try:
        y=7/0
except ZeroDivisionError as e:
        print ("div-by-zero happened: ", e)
```

    - we can catch multiple exceptions at once with a tuple of them:

+ EX:
```
try:
        y = float(eval(input(" number please: ")))
except (ValueError,TypeError,NameError) as e:
        print ("uh-oh!", e)
```

    - we can also have multiple except-blocks (unlimited). Only 1st appropriate one is run, others skipped.

+ EX:
```
try:
        y = float(eval(input(" number please: ")))
except ValueError as e:
        print ("bad value: ", e)
except TypeError as e:
        print ("bad value: ", e)
```

- else: only runs when the try-block was successful. (Exceptions might be generated inside an else-block, so it *does* make a difference between placing code here and code at the end of the try-block)
  + EX:
```
try:
        y = float(eval(input(" number please: ")))
except Exception as e:
        print ("sadness...", e)
else:
        print ("hooray, no exceptions!")
```

- finally: always runs: whether try-block was successful, or an exception block handled some exception, or even when an exception is propagating! Good place for 'cleanup' code that would've been in many places.
  + EX:
```
try:
  y = float(eval(input(" number please: ")))
except ValueError as e:
        print ("value error: ", e)
finally:
        print ("I always print, for no-, deferred- and handled-exceptions!")
```
always executed

-

- Defining Our Own Exceptions:
  - define a new class for our exception. Inherit from the Exception class, add attributes as desired.
    + EX:
```
class MyExceptionName (Exception):
        def __init__ (self, p1, p2):
                self.p1 = p1
                self.p2 = p2
        def __str__ (self):
                return (  "MyExc: %s (%s)" % (str(self.p1), str(self.p2))  )
```

  - now we can create objects of this type, and raise them.
    + EX:
```
if height < 5:
  the_exception = MyExceptionName("height too small",height) # create (call constructor)
  raise the_exception # raise the exception.
```

- Raising Exceptions
  - explicitly causes exceptional control flow by raising an exception object that seeks some try-except block that can handle it. (Example just above, in "Defining Our Own Exceptions")
  - creating an object of some exception type isn't enough, we must also raise it:
    +EX: `raise Exception("problem!")`

exception expression

`except TypeError as e:`
`raise e`

don't always have to catch
or
catch, handle, raise

- Deferring Exceptions
  - don't catch the exceptions. (So really, happens by default)
  - exceptions will be passed up the calling chain
  - deferring some: simply don't catch all Exception types that may occur, just some of them

- Handle and re-raise exception
  - Raise an exception in the except-clause
  - Handle and still propagate so that other exception handlers might see the exception

- Validating User Input:
  - we should never assume the user gives valid input – always check. correct input needed
  - try-except blocks can recover from serious user errors.
  - only loops re-try code; if you need to re-try something, the try-except block should be in a loop.
  - +EX:
```
def get_int (msg):
    while True:
        try:
            return int(input(msg))
        except:
            print("oops, try again:")
```

Recursion

- Recursion implies something is defined in terms of itself.
- We write functions that call themselves recursively.
- Approach:
  - identify **base cases**, where we can calculate the function's answer without recursion.
    - a function might have multiple base cases.
  - identify **recursive cases**, where the function calls itself on a smaller version of the problem.
    - 'smaller problem' means progress towards a base case has been made. Often an integer value changes towards the base case values, or a list's length shortens, or some measurable quantity approaches the one identified in the base case(s).
- handle base cases first to avoid "infinite recursion" (akin to an infinite loop).
- EX:

```
def even(n):
    # base case
    if n==0:
        return True
    # base case
    if n==1:
        return False
    # recursive case on smaller problem
    return even(n-2)
```

- Semantics: each recursive call is its own frame on the stack; this means each call has its own copy of local variables, and each one separately returns its own value to its own specific return location.

- Pros:
  - each call has local variables/scope
  - often looks like the mathematical definition. We already define things recursively by nature.
  - can handle irregular shapes (like the sum_list example from slides).
- Cons:
  - uses more stack space – might get stack overflow!
  - function call takes slightly longer than jump to next iteration.

- When is recursion an acceptable approach?
  - if # recursive calls is minimal in relation to size of input, and code runs 'fast enough', then recursion is a decent solution. See slides for specific examples of good and bad times for recursion.