CIS 480 Exam #2 Review Suggestions

* last modified: 11-03-05, 11:43 am

* remember: YOU ARE RESPONSIBLE for course reading, lectures/labs, and especially anything that's been on a homework, in-lecture exercise, or lab exercise; BUT, here's a quick overview of especially important material.

* you are permitted to bring into the exam a single piece of paper (8.5" by 11") on which you have handwritten whatever you wish. This paper must include your name, it must be handwritten by you, and it will **not** be returned.

  Other than this piece of paper, the exam is closed-note, closed-book, and closed-computer.

* this will be a pencil-and-paper exam, but you will be reading and writing Python code, statements, and expressions in this format.

  Note that a packet of Python code **may** be included along with the exam, both for reference and for use directly in some exam questions --- the ability to make use of existing code as a reference is a vital skill in Python (as in most programming languages).

  * note that you could be asked to write Python expressions, statements, functions, or up to and including entire Python modules;

    (note, too, that answers may lose points if they show a lack of precision in terminology; for example, if I ask for a literal or an expression and you give an entire statement, instead)

  * note that I could ask you questions *about* Python, or about various aspects of Python;

  * note that I could ask you what given Python code does or means; I could give you one or more statements, a function, etc., and ask you what it does or what it would output in a given situation (or how you could write a call to use it, etc.)

  * you could be asked to modify a piece of code or function or module, or to correct a segment of code or a function or a module, as well;

* **Tuples**

  * You were responsible for tuple basics last time --- now I'll expect you to be comfortable writing code involving them, too.

  * How do you write a 1-item tuple literal?

  * I will **not** ask about when you don't need parentheses around a tuple --- since I think it is better style just to put them in, for readability.

* **Dictionaries**

  * Likewise --- you were responsible for dictionary basics last time, but now I'll expect you to be comfortable writing code involving them, too.

  * What are the practical differences between grabbing a key's value with the [ ] notation, and grabbing it

with the dictionary **get** method?

- With regard to dictionaries: what can you do with a tuple that you cannot do with a list? Why?
- make sure you can iterate through a dictionary in a variety of fun and useful ways; make sure you can read code that does so, also.
- Be comfortable reading/writing dictionaries being used to implement multi-way branches (see Learning Python, Ch. 9, pp. 147-148, as well as the Week 10 Lecture and Lab, going deeper into functions).

* **Files**

  - What does the **open** function return? What are the three processing modes possible for **open**'s second argument, and what do they mean?
  - Should be able to read from a file --- be comfortable with at least the following file methods:

    read(), read(N), readline(), readlines()

    - what do these return if called after the "end" of a file stream?
    - why might you sometimes want to use **xreadlines()** instead of **readlines()**?
  - Should be able to write to a file --- be comfortable with at least the following file methods:

    write(S), writeline(L)
  - In terms of course style guidelines (and occasionally even for practical reasons), what are expected to do to a file object when you are done using it? Be able to write the statement for doing this, too.
  - How can you do interactive input in Python?
    - you are only responsible for the **raw_input()** function; there will not be any questions on the **input()** function, as its use introduces potential security issues.
    - Be confortable reading and writing code making use of **raw_input()**

* **Focus on TYPING in Python**

  - In Python, a variable [Learning Python, p. 69] "**never** has any type information or constraint associated with it"; where does the notion of **type** "live", instead?
    - what are the implications of this for Python variables?
  - What must be done to a Python variable before it can be used?
  - I may ask questions to see if you are really getting the distinction between names and objects;
  - You may be required to read/write name/object "pictures" liked we used on the whiteboard in lecture to describe/show what is really happening as different assignments are made.
  - What happens in Python to a non-integer, non-small-string object once no name is referencing it?

- It is quite likely that I will give you some code, and ask you what the involved variables' values are after that code (which change? which do not?); I could also ask you why.
- How do assignments differ when mutable objects are involved vs. when immutable objects are involved?
- What are the three type (and operation) categories in Python? Be able to give examples of type(s) in each.
- How can you get a **copy** of a mutable object, as opposed to a reference to it? How can you get a "**complete, fully-independent** copy of a **deeply-nested** data structure" [Learning Python, p. 120]?
- What does == test for? What does **is** test for? Be comfortable reading, writing both, but also be aware which one is more commonly used.

* **Intro to REGULAR EXPRESSIONS**

  - which module did we use for these? (There's more than one, but this one is the only one that we discussed, and is the only one you are responsible for.)
  - Remember that you were assigned to reading the Kuchling tutorial for this; (and, the link to it is still available from the public course web page).
  - What are Regular Expressions (RE's)? How can they be used?
  - How do you use **re** to obtain a regular expression object? What function do you use? What argument(s) does it expect? What does it return?
  - Let's say you have a regular expressions object. You should be comfortable reading/writing code involving the regular expression object methods **search**, **match**, **findall**.
  - Since several of the regular expression object methods return **match objects**, you should also be comfortable reading/writing code involving the match object methods **group**, **start**, **end**, and **span**.
  - You should be comfortable reading/writing code involving the following RE metacharacters: **[ ] \ . * + ? { } | ^ $ \b \B \A \Z**
    - In terms of grouping (using **( )**), you are only responsible for knowing that you can use them to apply something such as * to a collection of items;
    - how does ^ behave when it is the first character inside **[ ]** ? How else can it be used?
  - You should be comfortable reading/writing code involving the following RE predefined special sequences: **\d \D \s \S \w \W**
  - Remember: in what form does one usually write regular expression strings? Why?
  - What do we mean when we say that repetitions such as * are "greedy"?
  - Be comfortable with the re compilation flags re.IGNORECASE and re.MULTILINE

        * how do ^ and **$** behave differently when compiled with re.MULTILINE as when not?

    * Chances are extremely good that you will have to write, read RE strings (and other RE-related code)

* **Deeper into FUNCTIONS**

    * What are the implications of a **def** statement actually being executable code? (What does that mean you can do in Python that you cannot do in Java/C++?)

    * When Python reaches and runs a **def** statement, what really happens? (What is generated, what is assigned?)

    * If you assign a name to a function object --- how can you call that function object using that name?

    * What is a **higher-order function**? How can you write one in Python?

        * Be comfortable reading/writing functions that take functions as arguments; be comfortable reading/writing functions that return functions as their results.

        * Be able to call a function that takes a function as an argument; be able to call a function that returns a function as its result (and call that returned function, too).

    * (as already mentioned, this lecture/lab is where we discussed using dictionaries to implement multi-way branches...)

    * Should be comfortable reading/writing Python **lambda expressions**, and code involving them;

        * What is a **lambda expression's** value? What is its syntax and semantics?

        * How does a **lambda expression** differ from a **def statement**?

        * What are its limitations? Where is it particularly useful? Where can you use a **lambda expression** that you could not use a **def statement**?

    * Be comfortable reading, writing code involving the built-in functions **apply**, **map**, **filter**, and **reduce**.

    * How are parameters passed in Python? How does this differ from parameter passing in C++? (how does it differ from **pass-by-value** and **pass-by-reference**?)

        * What kind of Python arguments, in practice, usually end up "behaving" like C++ pass-by-value?

        * What kind of Python arguments, in practice, usually end up "behaving" like C++ pass-by-reference?

    * Be comfortable with Python **scope** rules;

        * You should be able to read a code fragment (or several files' contents) and determine the **scopes** of the names within them;

        * What is the scope of a **def** statement's parameter names?

        * What is the assumption when a name is assigned within a **def** statement? How can you "override"

this assumption, if you want to?

- What does so-called **global scope** *really* mean, in Python? What does it really span?
    - However, how **can** you use a "global" variable elsewhere?
- What does **LEGB** mean? (what is the **LEGB** rule?)
- Name references in Python search at most **four** scopes --- what are they?

* You should be comfortable reading/writing code using the four **special argument-matching modes** discussed;
    - How can you write a function that can take a **varying** number of arguments?
    - How can a parameter have a default value if no corresponding argument is provided?
    - What are **keyword arguments**? How can you use them in a function call? What are some of the potential benefits of using them?
    - How can a function be written to grab **unmatched keyword arguments** and gather them into a dictionary?

* **Focus on Modules**
    - You are responsible for the material covered in the Week 11 Lecture (11-1-05), but I will keep in mind that you haven't had a homework covering this material yet. (You will have by the Final Exam...) You should be comfortable with the basic concepts discussed.
    - ...although you should be VERY comfortable with basic module use, since we've been using them all semester, and you should be VERY comfortable with the **import** statement;
    - what does the **from** statement do?
        - Even without having had a homework on this, from the lecture and your reading you should be able to read, write **from** statements;
        - How does **from** differ from **import**?
    - what does the **reload** function do?
        - Even without having had a homework on this, from the lecture and your reading you should be able to read, write **reload** function calls;
        - What argument does **reload** expect?
        - Why is **reload** needed? (What can it do that **import** cannot?)
        - Can you **reload** a module which you have accessed using just **from**?
    - What are the large collection of utility modules automatically provided by a Python standard installation called?

    * How can you access and use them?

    * Currently, roughly how many of them are there?

    * Where is a good source for browsing through documentation on them?

* How does an **import** statement differ from a C++ **#include**?

* What are the **three distinct steps** that are done the **first** time a module is imported by a program?

    * When you have used the **import** statement with a module name --- what happens as a result? (What name or names are assigned? And what do they refer to?)

        * What implications does this have on how you can name your Python module files?

    * Whe you have used a **from** statement (or **from ***), what happens as a result? (What name or names are assignmed? And what do they refer to?)

* How does Python locate the module file referenced by an **import** or **from** statement?

    * What four pieces are concatenated to compose its search module path?

    * What is **PYTHONPATH**? How can it be set up in a bash shell such as cs-server? What would its effect then be on the search module path?

    * What is the **sys.path** list? How can you look at it? What would happen if you appended to it? How does Python use it?

    * What is the practical reason why Python is designed so that you do not put the module file suffix in **import** and **from** statements?

* When is a **.pyc** file created? What is in it? How can Python use it? What is its benefit?

* What is the **sys.modules** dictionary? How can you look at it? How does Python use it?