Introduction to Computer Science CS105

Lab 6: Making Unconventional Change

Text in red was added later, to clarify

Deliverables

Commit and push deliverables 1a, 1b, and 1c on/before 11:00 pm, Tuesday, October 7, 2025 and ~~the remaining elements on/before 11:00 pm, Thursday, October 23, 2025~~
The rest of Part 1 is due Sunday, October 26, and Part 2 on Tuesday, October 28.
Metadata: Your `partnerAndResources_Lab4.md` and `AIUse_Lab4.md` files, in MakingChange

1. All elements for the core wackyChange algorithm (with lab partner)
    a. Precondition expresssions in python for `wackyChange` and `wackyChange_012`
    b. Postcondition expressions for `wackyChange` and `wackyChange_012`; write as much of this as you can in python, and add comments for anything else
    c. A design document for `wackyChange_012`, exploring *several* possible design strategies, and evaluating them
    d. A python function to solve the problem
    e. Additional tests, as needed
2. An updated text-based user interface for your makeChange and wackyChange functions (with lab partner)
3. There is *no* individual programming work for this lab.

Make sure you ***commit and push*** your final work before the deadline, to get credit. We recommend you commit (and, if you like, push) frequently. If you feel you need to revise your initial submission (1a, 1b, 1c) after the initial deadline, do so: the initial submission should correspond to some sensible potential answer, but your final submission of those elements should correspond to the function you wrote.

Introduction

Review the basics of the change-making and wacky change problems (here).

Although this lab seems similar to the previous change-making lab, the subtle differences can sometimes have a big impact on what algorithms and programming idioms will work. So, keep in mind the principle that picking the right design and programming technique will be a major determining factor in how much work you do and how successful you are. Having thought about the *question* by making a good test suite, with your previous partner, will help, but don't let that substitute for combining a good set of tests with careful thought about design.

As always, keep in mind

- the Collaboration Policy, which means you may discuss ideas about this lab with anyone, as long as you don't get/view python code from someone/something other than your lab partner (or give out your code), as long as you mention your collaborators in the partnerAndResources file.

So, feel free to share *tests* and design *ideas* with your classmates, even if they're in different groups, as long as you document that you did so.

- The AI Use Policy: You can use cs50.ai if you want to, but we *strongly recommend* you only ask it if you've already spent some time working and are really stuck. Record your use in the AIUse file, as discussed in Lab 1 and the policy document, including any responses you find really *good* or really *bad.*

0. Set up to do your work, enter lab partner information

Open the **MakingChange** project. For Lab 5, you should have a lab partner, but this time you are welcome to work again with someone you've worked with *once* before, or someone new, but please don't work with someone if you've already worked with them twice.

1. Solving the wackyChange problem (with your partner)

In the **MakingChange** project, create files `wackyChange.py` and `wackyChange-Design.md` in which you'll create and discuss the design of a function `wackyChange` that solves the *general* change-making problem. This problem, and, thus, the function you'll write, **differs** from the previously-assigned `makeChange` function in several important ways:

1. **Its parameters:** This function should accept a *list* of denominations as a 2nd parameter (after the amount of money to be given). So, for example, the function call

   ```
   wackyChange(354, [100, 50, 20, 10, 5, 1])
   ```

   would be analogous to the $354 test from the original `makeChange` function that used U.S. currency $100, $50, $20, $10, $5, $1.

2. **Its return:** This function should return a *list* of how many of each bill to give, where, e.g., the first element in the returned list would be the number of bills to include of that first denomination in the denominations list. So, for example, the result

   ```
   [3, 1, 0, 0, 0, 4]
   ```

   would indicate that we are to give three $100, one $50, and four $1 bills, when asked to produce a total of $354 with the fewest bills.

3. possibly **its design:** Changes to the details of a problem sometimes call for a re-design; that may or may not be important, in this problem, depending on what design approach you used for the previous lab.

- **Remember** that your goal is to provide the smallest *number* of bills, so, you can't produce $350 as three $100 and fifty $1 bills (or as seven $50, etc. … you need three $100 and a $50).

It should be possible to use your function for ***any set of distinct positive integer denominations***, but your algorithm doesn't have to work for a $ -2 bill, or a $ 3/7 bill, or a $ π bill. If you wish, you may also require that the list of denominations includes the value 1 (rather than, say, having $2 and $5 as the smallest two bills, dropping the $1 bill, and making it impossible to make change for $3), or require that the list of denominations be given in some specific order, but these assumptions must be checked in the **precondition** (using Python code, if possible).

- Create **appropriate tests** for the `wackyChange` function, demonstrating a variety of parameters and results. Include some *familiar* systems, e.g., some that worked in makeChange. Also some *unfamiliar* ones, demonstrating different currency choices. Note that examples involving large numbers, e.g., over 360, might run slowly if you are using recursion, so you can omit such tests from your test suite. You should start with your tests you wrote in the previous lab, and discuss these with your lab partner, adding any other tests you like, from their test suite. Then, add more tests as you see fit, as you work on the lab.

- Include a **precondition** that distinguishes legal from illegal values of the parameter values. Don't just rule out all the interesting cases such as the $7 and $17 bill, as *it must be possible to use your function for any set of distinct positive integer bills that include 1*, but you should rule out inappropriate cases such as asking for - $28 in change, or giving you a list of bills containing only a $100 bill.

  You should be able to write your precondition as a python expression; feel free to either use standard python library functions or write a brief helper function. If someone calls your function with an illegal parameter (such as asking for - $28 or allowing only the $100 bill), your precondition expression should evaluate to False, causing the precondition function to throw an exception, indicating an error in the code that *called* your function.

- Include a **postcondition** that distinguishes a correct from (if you had one) an incorrect return. This should rule out any illegal result if you were asked a legal question, e.g. when asked for $354 in change with the six most-common U.S. denominations, and would produce four $100 bills, or only four $1 bills and nothing else.

  If you can, write your postcondition as a python expression, possibly using library functions or a brief helper function. If you do so, your postcondition expression should evaluate to False for any illegal result, causing the postcondition function to throw an exception, indicating an error in your code. You should be able to check at least *part* of the precondition as a Python expression, but you can include other elements as comments; If you're not able to write any of the

postcondition as a Python function, write it all as a comment, for 80% credit on this step (if your comment is complete and correct). It's also fine to write one or more other functions to do the check (hopefully they'll be simple).

If you write your postcondition before the code, which is often a good approach, note that it sometimes helps to just create a variable "answer" (or whatever name you like), initialized to some sort of empty value, and then check that. The check will, of course, usually fail until you actually write your function.

- Evaluate possible **design approaches**, including at least "Design By Cases", "Basic Recursive Design", and "Imperative Design" (i.e., think about what you'd do to solve this problem, including updating the values of variables/objects), and, if you like, any hybrids or other designs you'd care to consider, such as combinations of top-down and basic-recursive designs, or combining basic-recursive design with a loop, etc…

  Put this discussion in a file wackyChange-Design.md. Provide some detail about each approach, not just an evaluation "that would work" or "that wouldn't work" … give a sense of how you would be able to use this approach or, if you think it's not applicable, *why*. If you envision a viable design by cases, what are the cases; for a viable loop-based design, explain what variables you'd be updating and how you'd update them; or, for a recursive design, what your smaller problem(s) would be, and how you'd use their results.

  For your designs, it's o.k. to consider just the "zero-to-four" simplified version of the problem, i.e., you can assume you don't ever need to give more than *four* of any bill.

  You should include at least one design you think *will* work; you may choose any workable strategy that does not involve making changes to/with non-local variables[1].

- Create the first line of the function definition, with appropriate parameters, including the parameter and return types. Remember that you need to use "import" to get access to the type List, i.e., use

  ```
  from typing import List
  ```

  before writing something like `List[str]` or `List[int]` or `List[float]`.

- Write code for your **algorithm** as your python function; you'll need to think carefully about whether the algorithm for the makingChange lab will work for this more-general case. It relied on assumptions about the standard American currency, and may need major modification, or

[1] The phrase "changes to/with non-local variables" means having a function *f* that either makes a change to a *variable* that's neither created within *f* nor one of *f*'s parameters, or (b) using such a variable to make a change to an object it refers to. It's o.k. to have "global variables" that don't *change*, if you like, and if you're using the imperative approach you can change parameters and local variables as much as you want/need to. But, for CMSC 105, avoid making changes to global variables, since that tends to complicate reasoning about code no matter which paradigm we use.

you may need a completely different approach. You may use recursive function(s), loop(s), or neither of those. See the end of this document for hints and suggestions.

- If your `wackyChange` function accepts currency lists in any order, you'll need to make sure your output matches, i.e., if you allow `wackyChange(54, [20, 5, 50, 1, 10, 100])`, you need to indicate that you'd give one $50 and four $1 by producing the list `[0, 0, 1, 4, 0, 0]`, but not `[0, 1, 0, 0, 0, 4]`

- Make sure your code is **clear** to a programmer who might read it. Remember that good variable and function names, good organization, and good comments, are all tools to use to make your program clear.

See ** (below) for partial credit hopefully-easier versions of the wacky change problem, or the tiny-font hints below that, if you like (to read those, copy-paste them to another file/window to use a different font size).

2. Update your text-based user interface (with your partner)

Improve your UI (or create a new wackyChange_UI) to allow use of the above `wackyChange` function, including entry of valid sets of currency.

1. Confirm that you can run your program by typing the following at the command line

   ```
   python3 makeChange_UI.py (or wackyChange_UI.py)
   ```

2. For this lab, try to make it easy to both enter the currency and then ask for a variety of amounts of change using that currency, and then, if you like, you can let them switch to a different currency without restarting the program. You should also be careful that unexpected inputs from the user don't crash your program (e.g., exiting due to a precondition failure or other exception)... print a nice error message that a non-programmer could understand. And, of course, print the amount of each bill to give out, if the input *is* valid (the output can look like the regular MakingChange lab, but of course the currency will be whatever was entered, and it will be printed in your *user interface*, not the function that does all the work). For Fall 2025, note that we'll be very flexible about the appearance of the output, since no details were stated here in the original lab, and thus any reasonable output will be accepted.

3. Submit your work, if not already done (instructions)

Preliminary more-concise hints/suggestions for the wacky change calculation:

Before you program, be sure to write out at least one fairly detailed design description with a couple of steps of an example to show how it works (as described in Lab 4).

If you write multiple functions (not every approach involves these, but some good approaches do, so don't worry if you *did* or *didn't*, neither one is necessarily a problem), have tests for each function, including a simple test or two, as well as complicated ones.

It is often easier to solve a simplified version of a problem, and then return to the main problem … the biggest difficulty with that is that you need a lot of experience to guess what simplification won't totally throw off your algorithm. If you're having trouble figuring out how to program the general wacky change problem, and you've already done the *much* simpler making change from Lab 4, here's an intermediate one that *will help you develop an approach that can work for wacky change*. So, try solving:

The "0/1/2 wacky-change problem". In this variant, your cash register has only *two* of each bill, except for an unlimited number of $1 bills. So, if asked for $304 in change, you *can't* give three $100, because you only have two, and you end up giving two $100, two $50, and four $1. Write some tests for which the best answer doesn't involve having more than two of any given bill, and (for now) comment out or ignore any tests where you *do* need three of some bill, and write a function that only handles this case where there are two of each bill. **Fully test this function**, then, once it's working, we recommend you do a "commit" so that you can refer back to it later if you want to, and *then* modify that function to get rid of the "only two of each bill" limitation.

Note for 2025: the purple version above can take the place of *both* the 0/1 version and the zero-to-four version described below, and I plan to remove those for next year, since this one example brings together the best of each of those two.

Hints and suggestions

See design suggestions in tiny text below Start with simpler variants of the problems, e.g., assuming you'll only need to give one bill of each value (or none) rather than multiple bills of the same value; then, consider the problem if you never need to give more than four bills (which is the case when you're using the U.S. currency and don't have to give over $499)

**** Zero-One Version of Wacky Change.** If you're struggling to find an algorithm that works, consider solving the easier-to-figure-out "zero-one" version of the problem. In this version, we only have *one* of each bill except the $1 bill (which is unlimited). So, I can choose to give out the $100 or not, and I can choose to give the $50 or not, and I can choose to give the $20 or not, but I can't give out more than one of any of them. So, if we need to provide $90, we don't give out the $100, we do give the $50 (so only $40 more is needed); we'd *like* to provide two $20 bills, but in the "zero-one" version of the problem, we only have one $20, so we give $20 + $10 + $5 + five $1 bills. Remember that you still should give the minimum number of bills, so no fair answering the request for $90 with ninety $1 bills.

The "zero-one" version should be easier to design and program than the full wackyChange, and *usually* leads students to a design that can be generalized to the full problem. Solving that "zero-one" version of the problem will get you a large fraction of the credit for this lab. So, if you're stuck, try solving that, then do a "git commit and push", then try to do the version where you have unlimited bills, and commit and push that if you get it working.

** Zero-to-four version. Like the zero-one version, but you can give out up to four of any given bill, rather than just giving one of them or not; *while working on this, don't worry about repeating some code several times in the body of your function*; doing so is normally considered bad style, but sometimes the best way to solve a hard problem is to do something that's stylistically poor and then refactor that away once you get the system working. This restriction won't work for the general problem, e.g., giving change for $343 using only $7 and $1 bills, but if you can solve this, and then improve any style issues having to do with "writing essentially the same code in multiple places", you should then have code that can easily be edited to handle the general case. If you hand this code in, even without the "improve any style issues", it will be worth 90% of the credit for programming (if it's right).

Hints/Suggestions for designing “wacky change” (copy to another document to view)

Clearly, any ordinary makeChange function that had six "if" clauses, each dealing with one bill, **can't** easily be restructured to work with a variable number of currencies. But even loops and recursive algorithms can contain assumptions about the currency that may no longer be valid (such as the relationship between the bill values).

So, come up with some really weird currencies for your tests, and ask yourself: will the same basic approach to making change work? It can also help ask "How do *I know* that the approach for the previous makeChange algorithm *must give the fewest possible bills*?". Or, to be even more specific, in order to make change for $354 using the standard U.S. currency, we should give *three* $100 bills. Can you explain *why* any algorithm that gives two $100 bills must be wrong?

More precisely, the previous algorithm for making change with standard US currency uses what is called the "greedy method", where each bill, from highest to lowest is maximized, and that gives the fewest possible number of bills. This approach may not give the fewest possible bills for arbitrary currencies, so come up with examples to check this.