CS 112 – Project Basics

Each project has some similar characteristics: the general flow, requirements, and so on. This document records the similarities for all of our assignments, so that we don't have to keep repeating ourselves in each project.

Individual Effort

All projects in CS 112 are individual efforts. You may only contact course staff with questions about the projects. Any use of websites, discussing with other students, attempting to copy, share, buy, steal, or otherwise acquire a solution other than self-creating it on your own will result in an honor code case with recommendation to fail the course plus further measures. *It's stupidly easy to compare project source files for text similarity!* Just don't do it. By the time a project is assigned, you'll have read on the topic; seen it in lecture; and worked on the ideas on collaboration-allowed lab work. If you still have questions, that's what private piazza posts and office hours visits are for. You're not stranded alone! You just need to go through the appropriate channels and everything is 100% fine.

Deadlines, Tokens

The project lists a deadline, probably 11:59pm on a particular day. Late work is accepted up to 48 hours late. Whether there is a grade-reduction or spent token is automatically determined as follows:

- The last submission you make is the only one we grade, automatically. (any turned in after 48 hours are ignored though!)
- If your work is turned in 0-24 hours late, it's considered one day late; if it's turned in 24-48 hours late, it's considered two days late.
- Any One-Day-Late tokens you have are automatically applied to reduce the number of days late.
- Any days late that couldn't be covered by tokens (e.g., you'd spent them all) each result in lowering the highest possible score you can get by 25%. This is better for you than simply deducting 25%. Think of it as calculating your raw score (before deadline considerations are made), and then you'd get:
    - one day late: recorded_grade = min (75, raw_score)
    - two days late: recorded_grade = min (50, raw_score)

If you're running the tester and see that you've currently got about a 43%, and then work one day late to get up to a 73%, you'll still get a 73%, hooray! (Note that if any tokens remain, they get spent for late work no matter what grade you get).

Getting Assistance

If you have questions, use the Piazza discussion forums (and professor/TA office hours) to obtain assistance. *Remember, do not publicly post code for assignments on the forum!* Ask a general question in public, or ask a private question (addressed to “instructors”) when you're asking about your particular code. Also please have a specific question; instead of "my code doesn't work, please help", we need to see something like "I'm having trouble when I add that particular line, what am I misunderstanding?". If you are unsure whether a question may be public or not, just mark it as private to be sure. We can change a post to public afterwards if it could have been public.

Requirements

In each project, you will turn in a Python file on BlackBoard to the appropriate assignment, using our naming convention of:

`userID_2XX_KN.py` example: *`gmason76_230_P1.py`*

But you must use:

- your actual userID (like gmason76)
- your LAB section number (201 – 230, or 2B1-2B8 in the summers) instead of 2XX
- `P` for projects, `H` for homework, `L` for lab work, and then the assignment number, e.g.: `_P1.py` , `_H2.py` , `_P4.py`

Notice the use of underscores, not dashes. If you misname your file, you'll probably lose points when we have to spend extra time finding where your file went in our bulk download/sorting-by-name.

Procedure

Each assignment will have you download a testing file, linked like the following examples **(which may or may not exist at the moment – just look for these kinds of links at the top of the specification documents):**

- Download this tester file: http://cs.gmu.edu/~marks/112/projects/tester1p.py
- Download this tester file: http://cs.gmu.edu/~marks/112/labs/testerL2E.py
- …

Early on, we might also provide a template (something like p1template.py). Whether you download that and rename it, or just start your code file fresh, this is the only file you actually need to write code into. Remember to name it by our convention (e.g. gmason76_230_p1.py), and put function definitions inside of it as required.

Downloading Files

You should download an exact copy of it. Hopefully you can just follow the link, "save as…", and save the file. But you might have settings from your operating system that don't trust downloading code, or it might hide the extension while also enforcing another non-.py extension (Windows is the usual culprit for this one). That would mean you've got a file named tester1p.py.txt, or tester1p.py.htm, or some other weird name! If that ends up being the case, here are some ideas to get around it:

- figure out how to tell your computer to always show you extensions. (Google for solutions)
- copy all the code and paste it into a new file *inside of your code editor*, which is savvy about preserving spacing and (the utter lack of) formatting.
- try various browsers; sometimes Firefox is better than Edge, for instance. Usually Chrome works for me.

Where to put files

I'd suggest having a separate folder for the course, and perhaps even for each assignment. You should put all files for an assignment in the same place: the tester, your code, any other files required (occasionally there are data files, once we learn about reading/writing files). This will be your "working directory".

Editing Your Code

We have you define functions, and then we provide tests that will call your functions. It's important to note that your function is the "factory" that is designed to accept some inputs, do a calculation, and return a single output value. This means that it's the job of whoever wants to use your factory to actually supply starting values. Let's suppose our project asked us to define the add3 function: we probably end up with something like this in our code: (we will test this add3 function (and add4, and mul5 for extra credit!) below)

```
def add3(a,b,c):
    ans = a+b+c
    return ans
```

Navigating to your files

You need to open up either Terminal (mac) or the command prompt cmd (windows). This is like a chat window to your computer, but doesn't actually have anything to do with Python. Just like a file browser, it has a current directory. We need to navigate to where you put your code. Here's my example, getting into a folder on my desktop. On windows, the **`dir`** command does what **`pwd`** and **`ls`** do all at once. We all use the **`cd`** command.

```
demo$ pwd
/Users/mudd
demo$ cd Desktop/cs112/project1/
demo$ ls
gmason76_230_p1.py   tester1p.py
```

Running Code Interactively

First and foremost, you should be able to run your code and perform calls to your functions interactively. It might look something like this. It is useful for exploring the behavior of your code.

```
demo$ python3 -i gmason76_230_p1.py
>>> add3(1,2,3)
6
>>> add4(10,15,20,25)
70
>>> quit()
demo$
```

Running the Tester

You should also be comfortable running the tester file, which will stress-test your code with many different uses (feeding inputs, checking for specific output values). Though the exact number of tests will vary, each assignment will hopefully look like this when you've finished the assignment:

```
demo$ python3 tester1p.py gmason76_230_p1.py

Running required definitions:
....................
----------------------------------------------------------------------
Ran 20 tests in 0.001s

OK

Running extra credit definitions:
.....
----------------------------------------------------------------------
Ran 5 tests in 0.000s

OK

20 / 20 Required test cases passed (worth 5 each)
5 / 5 Extra credit test cases passed (worth 1 each)

Score based on test cases: 105.00 / 100 ( 20 * 5 + 5 * 1)
demo$
```

It ran all the required tests (20 here), and some extra credit tests as well. They all passed, so we see OK for both sections, and then some grading details. Hooray, we got 105%!

Understanding Tester Output

In reality, we probably don't pass all tests right away, and that's okay. We'd probably see something like this: we fail one test case (hence the F for "failure" instead of another dot for passing, or sometimes E for "error"):

```
demo$ python3 tester1p.py gmason76_230_p1.py

Running required definitions:
F...................
======================================================================
FAIL: test_add3_1 (__main__.AllTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "tester1p.py", line 118, in test_add3_1
    def test_add3_1  (self): self.assertEqual(add3(1,1,1),3)
AssertionError: 12345 != 3

----------------------------------------------------------------------
Ran 20 tests in 0.001s

FAILED (failures=1)

Running extra credit definitions:
.....
----------------------------------------------------------------------
Ran 5 tests in 0.001s

OK

19 / 20 Required test cases passed (worth 5 each)
5 / 5 Extra credit test cases passed (worth 1 each)

Score based on test cases: 100.00 / 100 ( 19 * 5 + 5 * 1)
demo$
```

I highlighted:

- where we can find the file and line number (the part of the tester where our code caused it to fail)
- what was actually being asserted as true (here, that calling add3(1,1,1) should equal the value 3)
- the specific issue that arose (here, that the value 12345 does not equal the value 3)

You should seek familiarity reading the useful parts of these messages. It's very challenging for a computer to both read your mind, and guess the purpose of your program (especially when it's wrong), in order to give good error messages; sometimes it seems cryptic, but it's giving you all the information it can about where it was and what it was doing when things fell apart.

narrowing the focus of testing

If you are overwhelmed by the 125 failed tests in a large project and just want to focus on one part, you can do so thanks to our feature-packed tester! When running the tester, precisely name the function(s) you actually want to test, and only the tests for those functions will be run.

```
demo$ python3 tester1p.py gmason76_230_p1.py  add4

Running required definitions:
..........
----------------------------------------------------------------------
Ran 10 tests in 0.000s

OK

10/10 Required test cases passed (worth 5 each)

Score based on test cases: 50.00/100 (10.00*5)
demo$
```

You can choose to run multiple functions' tests without running all tests by naming many functions, and it doesn't matter which ones are or are not for extra credit. Here, we test both `add4` and `mul5` at once:

```
demo$ python3 tester1p.py gmason76_230_p1.py  add4 mul5
```

Running Code in the Visualizer

Sometimes we know that we have a bug, but it's still hard to figure out where our expectations of the code and the reality of the code diverge. The tester helpfully told us some inputs that don't do as expected, but we are not yet sure why it behaves that way. The Python Visualizer is a great teaching tool to learn what's happening, and see your code run: http://pythontutor.com/visualize.html - mode=edit

Then, you should paste both your function and a specific call to it, e.g.:

```
# the definition:
def add3(a,b,c):
     ans = a+b+c
     return ans

# our chosen sample call, printed:
print(add3(5,10,15))
```

You'll be able to step through the code one line at a time, run backwards and forwards, see memory, and more. It's quite useful.

Tasks

Each assignment will list either some separate tasks, or functions you must define, usually with descriptions like the following:

- `add3(a,b,c)`: This function is given three integers. It calculates their sum and returns it. Here are some examples:

```
add3(1,2,3)         →     6
add3(10,-20,30)     →    20
add3(0,0,0)         →     0
add3(100,100,100)   →   300
```

You should either add this function to your file, or start editing the template (when provided on early assignments, to guide us). If we have a template, you're welcome to still write your own file – as long as the tester accepts it you're fine.

What can I use?

There is usually the question about what libraries are allowed to be used on projects. In general, Python (or the Python community at large) will often have modules available that seemingly have the full solution, already coded up! Obviously just calling these other libraries does very little for us to actually learn how to solve problems on our own, so they are in general not to be used in this way. Things mentioned in class, and in the project specifications, are okay unless explicitly disallowed by the project specification. If you find other options, you need to ask ahead of time if it's okay to use something. When we move on to harder topics, using libraries or built-in functions for the simpler things that we've already learned/practiced will slowly become okay as the semester progresses.

We will list what's allowed and not as needed, but when in doubt, ask first!
The rule of thumb is, calculate all the 'new' stuff by yourself, and don't import anything without permission.

This is one of the major differences between programming in a class and programming outside of a class. Our goal is the process, not the end product.

Commenting

If it's currently the interesting level of details on our project, it really needs comments. Each block of code, function, loop, branch, etc. should have a comment.

Comment your code sufficiently so that someone reading the source (the .py file's contents) is guided with clarifying comments and descriptions of each line of code. You might have upwards of half of the entire file be comments; this is normal (for well-documented code, anyways). Remember, comments should tell the whole story, and the actual code happens to implement that story. Please write comments as you go; it's much more useful to you than writing them all at the end. Also include the following comments at the very **top** of the file as follows, personalizing all information.

```
#-------------------------------------------------------------------------------
# Name: George Mason
# Project 1
# Due Date: 12/31/1999
#-------------------------------------------------------------------------------
# Honor Code Statement: I received no assistance on this assignment that
# violates the ethical guidelines set forth by professor and class syllabus.
#-------------------------------------------------------------------------------
# References: (list resources used - remember, projects are individual effort!)
#-------------------------------------------------------------------------------
# Comments and assumptions: A note to the grader as to any problems or
# uncompleted aspects of the assignment, as well as any assumptions about the
# meaning of the specification.
#-------------------------------------------------------------------------------
# NOTE: width of source code should be <=80 characters to be readable on-screen.
#2345678901234567890123456789012345678901234567890123456789012345678901234567890
#       10        20        30        40        50        60        70        80
#-------------------------------------------------------------------------------
```

Grading:

We will list out the breakdown of points on each assignment. Usually test cases are most of the grade, and then commenting (and occasionally submission procedure) are worth more points. Don't miss the easy points – comment your code and turn it in correctly, with the correct name!

```
Submitted correctly:          5
Code is well commented:      20
Calculations correct:        75
--------------------------------
TOTAL:                      100  + extra credit sometimes possible! ☺
```

Some things can go wrong on a project that don't directly fit our grading tables. They might incur penalties, such as:

- instructions said you couldn't call some particular built-in function, but you did (we might deduct all points on that part of the assignment).
- code couldn't be run without crashing, but we could quickly get it running and test it. You've already risked quite a lot by turning in broken code, but luckily your GTA came to the rescue and you only lost like 10-15%.
- code couldn't be run without crashing, and it isn't close enough for the GTA to fix it for running tests. The GTA will eyeball your code and try to award a few points back, but in this situation you are probably looking at 25-40% scores at best. *Turning in code that runs is a big deal!*
- commenting wasn't offered as part of the 100 standard points this time, but was still required; if you forgot, we might deduct up to 10%. Commenting code is a strongly enforced norm.
- something weird happened, like you turned in project 1 to the lab 1 folder, and made extra work for your grader. We will probably deduct points for this; please follow instructions!

What is "hard-coding"?

We do quite a bit of our grading via testers, which feed specific inputs to your code. There are often unlimited numbers of test cases; obviously we won't write them all down, nor will we spend the time testing them all. Your code ought to work on any legitimate inputs, but we only pick some examples that are representative of all the different situations we'd expect to be handled. "Hard coding" is when you write code that exactly checks for the test inputs we've chosen, and you return the known answer without computing it, trying to trick the tester file. Consider the following function that checks for prime numbers; it has some hard coding in it for sure!

```
def is_prime(n):
    if n<2:                # this isn't hard-coding; we are handling a range of
        return False       # possible inputs in a meaningful way.

    if n==5:               # there's nothing particularly special about 5. This block of code
        return True        # was added in response to a test case, so we've hard-coded!
    if n==1117:            # more hard-coding… we didn't calculate this, we "remembered" it from the tester.
        return True
    if n==2 or n==10 or n==12 or n==1000:    # we've enterprisingly grouped together a lot
        return False                         # of tests into one hard-coded bunch. Still bad!
```

If you hard-code test cases into your code, we will deduct at least the points that were tricked, perhaps more depending on the severity of it. Sometimes a specific value needs to be addressed separately, e.g. printing a list that has zero items in it; this doesn't count as hard-coding, because a solution that never looked at a testing file would still need to include that code. Only when we have directly added inputs-to-answer sections somewhere in our code is it considered hard-coding.

Reminders on Turning It In:

No work is accepted more than 48 hours after the initial deadline, regardless of token usage. Tokens are automatically applied whenever they are available.

You can turn in your code as many times as you want; we will only grade the final version (within 48 hours of the deadline). If you are getting perilously close to the deadline, it may be worth it to turn in an "almost-done" version about 30 minutes before the clock strikes midnight. If you don't solve anything substantial at the last moment, you don't need to worry about turning in code that may or may not be runnable, or worry about being late by just an infuriatingly small number of seconds – you've already got a good version turned in that you knew worked at least partly.

You can (and should) check your submitted files. If you re-visit BlackBoard and navigate to your submission, you can double-check that you actually submitted a file (it's possible to skip that crucial step and turn in a no-files submission!), you can re-download that file, and then you can re-test that file to make sure you turned in the version you intended to turn in. It is your responsibility to turn in the correct file, on time, to the correct assignment. You can always use a late token if necessary, but it's best to just review your submissions the first time around.

Back up your work!

→ Start with the first project, backing up your work somewhere safe. I'd suggest a cloud service like DropBox, GoogleDrive, and others. USB drives can be lost; we can accidentally delete the copy we're working on (but recover it with these services if we're careful!); so much can go wrong that you don't need to experience. Do yourself a favor and get in the habit now.