**General Debugging Tips + Tricks**

Most of this content has been graciously provided by CS 015, so you may have seen some of the content before. There are Python specific tips in this document, and a primer on how to use PDB!

**Debugging:**

Creating any program is a three step process: designing, developing, and debugging. In the design phase, you decide which features you need to implement and how to organize the classes in order to do so.

The majority of the coding happens in the development phase. Once you have testable code, you make sure it's working correctly in the debugging phase. This process is definitely not linear; often you will have to change your design after you begin coding, and you should always be testing your code during development.

Most of the time spent writing any program is in the debugging phase. A well thought out and thorough design will make development easier, and careful coding will simplify debugging tremendously. Nevertheless, every application is going to have bugs and being able to debug them is an essential skill.

**Incremental Coding:**

**Bad Coding:** Type all the code, try to compile it, get a hundred errors, run to a TA for help because you have no idea what's going on anymore.

**Good (Incremental) Coding:** Fill in a method and try to compile it. If it compiles, great! Now call the method. Didn't get the expected results? Look through the method, fix the bug and try again! Worked? Awesome! Fill in the next method. Rinse & repeat.

Good programmers will have bugs too. Incremental coding just ensures that when you run into a bug, you'll have a much better idea of where it is. Even if you can't figure out the bug, it will make it much easier for TAs to help you.

AI can be tough to code incrementally, since so many parts of the algorithm tie into each other. We recommend taking a bit of time before coding to think about how you can split the algorithm up into separate parts, and test them along the way.
**Strings and Printlines:**

Before talking more about specific types of bugs, we’re going to go over printlines, which will be helpful until you get access to more powerful means of debugging. Just like it is important to code incrementally, when there is a bug in your program, it is important to debug incrementally. You can do all of this using a built-in Python method, `print()`. This will print to your terminal whatever is inside of the parentheses.

Note that Python has some great formatting in strings which can help you format things easily. For example:

```python
>>> print 'test key: %s, test value: %i' % ('Hello World', 1)
Output: 'test key: Hello World, test value: 1'
```

For more info, you can look at the documentation:
https://docs.python.org/2/library/stdtypes.html#string-formatting

**Python and type errors:**

Since Python is a dynamically-typed language, you can run into some interesting bugs where you end up referencing a variable that is a different type than you initially expected. If you are running into errors, we recommend using the `type()` function, and tracking the type of the object through your program.

**Hand Simulation:**

This is a technique for debugging which focuses on looking carefully at your program and trying to find and fix bugs before they occur. There is nothing more frustrating than running your program over and over and trying to change little things until it works. It will take longer, and you often won’t understand what caused the error even if you solve it. Instead we recommend that you walk through all of the logic in your program before you run it, following the flow of control of the program from start to finish.

1. Check that variable names are correct.
2. Check that you are calling the correct methods at the correct places with the correct parameters.
3. Use a piece of paper to keep track of the value of each variable at each stage in your program.
Assert Statements + Unittest

Something that makes this class challenging is the fact that we do not give out huge test suites to make sure your code runs properly. Instead, that responsibility is on you! Python comes with a great testing library called unittest, which allows you to quickly and easily run test cases and identify where your program is going wrong!

Note: test functions within a Unittest class must start with the word test i.e:

```python
def test_bfs_order():
    pass
```

Unittests are powered by assert statements, which cover many cases. For example:

```python
self.assertEqual(a, b)
selfassertTrue(a)
assertIsInstance(a, b)
```

For a full list, visit:  
https://docs.python.org/2/library/unittest.html#assert-methods

Take a look at the documentation for more details about unit testing:  
https://docs.python.org/2/library/unittest.html

Getting Debugging Help at TA Hours

TAs are great for when you need a second pair of eyes to look at your code. Unfortunately, because lines are often long, we restrict students to 15 minute 1:1 sessions with a TA at a time.

To be clear, although you can expect TAs to know the algorithms and assignment specifications, you shouldn’t expect TAs to have intimate knowledge of your code or your specific approach to the problem. Maybe you used a Python set to keep track of data, whereas other valid approaches include a list or a dictionary! TAs won’t know this right away, and the more prepared you are to explain your approach, and the less time TAs have to spend learning your approach, and the more time you can actually spend getting help!

Suggestions for getting the most out of TA hours

1. **Explain your approach in words.** When your session with a TA begins, explain your approach to the problem. What datastructures did you use? Did you write helper methods?
2. **Comment your code** before coming to hours - what kind of datastructure does each variable represent, and what data does it keep track of? What does each helper function do? (something like the following would be awesome):

```python
>>> parents = {} #tracks parent-children, key = child, val = parent
```

3. **Use printlines!** Keeping printlines in your code (and commenting them with what each printline is meant to check) can be extremely helpful. These are great for showing the TA where things might be going wrong. However, try to only keep printlines that are relevant to your current problem, and delete those that are no longer needed, as extra printlines can make your terminal output hard to read.

4. **Pinpoint where your implementation fails.** Simply saying “my BFS doesn’t work” and handing over a chunk of code makes it hard for TAs to know what’s wrong, and sets you at a disadvantage to make use of the limited time you have together. Something like, “My BFS doesn’t work because my data structure isn’t keeping track of the visited nodes in the way I expect it to” is much more helpful.

In general, the more detail you can provide, the better. Remember that TAs aren’t responsible for solving your bugs; if your 15 minutes are up, they will need to move on to the next student, regardless of if the problem has been fixed. If you follow the above suggestions, you’ll make the most of your time, and chances are you’ll even be able to solve your own problems (and therefore won’t even need to wait in line at hours!)

**Quick Look at Using the Python Debugger**

In the context of this class, the general debugging techniques detailed above are usually sufficient. However, if you ever run into a situation where:
- lots of changing local variables
- large call stack
- you’re checking a specific sequence of execution
… or print statements are just too tedious to write

You always have the option to use the Python debugger (pdb)! You may have seen GUI (graphical user interface) debuggers bundled up with IDEs for languages such as Java - there are comparably Python IDEs ([https://wiki.python.org/moin/IntegratedDevelopmentEnvironments](https://wiki.python.org/moin/IntegratedDevelopmentEnvironments)) - but the easiest way to use the debugger is by running pdb in your source code.

The first thing you have to do is set a breakpoint in your python code. Whenever python hits a breakpoint, it'll interrupt the execution of the code and give you the ability to inspect the state of your program.
**example.py**

```python
def add_two_numbers(num1, num2):
    return num1 + num2

num1 = 10
num2 = 15
import pdb; pdb.set_trace()
total = add_two_numbers(num1, num2)
print total
```

**Tip:** if you're setting more than one breakpoint in your program, you can also `import pdb` at the top of your program and write `pdb.set_trace()` in multiple places.

Run your python code like you normally do, but this time, your program will stop at the breakpoint you set. In our case, python will halt execution when you're in the `add_two_numbers` method. It'll look something like this:

```
Jessica- Macbook-Pro:Desktop jessicafu$ python example.py
> /Users/jessicafu/Desktop/example.py(7)<module>()
--> total = add_two_numbers(num1, num2)
    (Pdb) p num1
10
    (Pdb) p num2
15
    (Pdb) n
> /Users/jessicafu/Desktop/example.py(8)<module>()
--> print total
    (Pdb) p
```

Execute the next line of code:

```
> /Users/jessicafu/Desktop/example.py(8)<module>()
--> print total
    (Pdb)
```

*This actually executes the current line of code that is displayed to you beforehand.*

Also keep in mind that if you try to execute a method call, the debugger will execute the statement and just pass you through to the next line of code. Instead, you can use the commands `s` to step into the method and `r` to get out of it.

**Check out the really good/gentle introduction to pdb that'll go over more interesting and useful things you can do with pdb:**