Debugging

> Adapted from cs61a of UC Berkeley.

Introduction

By now, you will have encountered various bugs when programming for this class. Most often, you will try to run your code and see something like this:

```
Traceback (most recent call last):
  File "<pyshell#29>", line 3 in <module>
    result = buggy(5)
  File <pyshell#29>", line 5 in buggy
    return f + x
TypeError: unsupported operand type(s) for +: 'function' and 'int'
```

This is called a *traceback* message. It prints out the chain of function calls that led up to the error, with the most recent function call at the bottom. You can follow this chain to figure out which function(s) caused the problem.

Traceback Messages

Notice that the lines in the traceback appear to be paired together. The **first** line in such a pair has the following format:

```
File "<file name>", line <number>, in <function>
```

That line provides you with the following information:

- **File name**: the name of the file that contains the problem.
- **Number**: the line number in the file that caused the problem, or the line number that contains the next function call
- **Function**: the name of the function in which the line can be found.

The **second** line in the pair (it's indented farther in than the first) displays the actual line of code that makes the *next* function call. This gives you a quick look at what arguments were passed into the function, in what context the function was being used, etc.

Finally, remember that the traceback is organized with the "most recent call last."

Error Messages

The very last line in the traceback message is the error statement. An *error statement* has the following format:

```
<error type>: <error message>
```

This line provides you with two pieces of information:

- **Error type**: the type of error that was caused (e.g. `SyntaxError`, `TypeError`). These are usually descriptive enough to help you narrow down your search for the cause of error.
- **Error message**: a more detailed description of exactly what caused the error. Different error types produce different error messages.

Debugging Techniques

Running doctests

Python has a great way to quickly write tests for your code. These are called doctests, and look like this

```
def foo(x):
    """A random function.

    >>> foo(4)
    4
    >>> foo(5)
    5
    """
```

The lines in the docstring that look like interpreter outputs are the **doctests**. To run them, go to your terminal and type:

```
python3 -m doctest file.py
```

This effectively loads your file into the Python interpreter, and checks to see if each doctest input (e.g. `foo(4)`) is the same as the specified output (e.g. `4`). If it isn't, a message will tell you which doctests you failed.

The command line tool has a `-v` option that stands for *verbose*.

```
python3 -m doctest file.py -v
```

In addition to telling you which doctests you failed, it will also tell you which doctests passed.

Usually, we will provide doctests for you in the starter files. You can add more tests by following the same format. It is often helpful to write additional tests to uncover more details about the shape of the inputs and the expected outputs of the problem, in addition to helping with the implementation of the program itself. A little time spent upfront writing tests can save a lot of time down the line.

Writing your own tests

In addition to doctests, you can write your own tests. There are two ways to do this: (1) write additional doctests, or (2) write testing functions.

To write more doctests, simply follow the style of existing doctests. You can also write your own functions (much like the `take_turn_test` function from Project 1).

Some advice in writing tests:

- **Write some tests before you write code**: this is called test-driven development. Writing down how you expect the function to behave first -- this can guide you when writing the actual code.
- **Write more tests after you write code**: once you are sure your code passes the initial doctests, write some more tests to take care of edge cases.
- **Test edge cases**: make sure your code works for all special cases.

Using `print` statements

Once the doctests tell you where the error is, you have to figure what went wrong. If the doctest gave you a traceback message, look at what type of error it is to help narrow your search. Also check that you aren't making any common mistakes.

When you first learn how to program, it can be hard to spot bugs in your code. One common practice is to add `print` statements. For example, let's say the following function `foo` keeps returning the wrong thing:

```
def foo(x):
    result = some_function(x)
    return result // 5
```

We can add a print statement before the return to check what `some_function` is returning:

```
def foo(x):
    result = some_function(x)
    print('DEBUG: result is', result)
    return other_function(result)
```

> Note: prefixing debug statements with the specific string `"DEBUG: "` allows them to be ignored by the `ok` autograder used by SICP. Since `ok` generally tests all the output of your code, it will fail if there are debug statements that aren't explicitly marked as such, even if the outputs are identical.

If it turns out `result` is not what we expect it to be, we would go look in `some_function` to see if it works properly. Otherwise, we might have to add a print statement before the return to check `other_function`:

```
def foo(x):
    result = some_function(x)
    print('DEBUG: result is', result)
    tmp = other_function(result)
    print('DEBUG: other_function returns', tmp)
    return tmp
```

Some advice:

- Don't just print out a variable -- add some sort of message to make it easier for you to read:

```
print(tmp)   # harder to keep track
print('DEBUG: tmp was this:', tmp)  # easier
```

- Use `print` statements to view the results of function calls (i.e. after function calls).
- Use `print` statements at the end of a `while` loop to view the state of the counter variables after each iteration:

```
i = 0
while i < n:
    i += func(i)
    print('DEBUG: i is', i)
```

- Don't just put random `print` statements after lines that are obviously correct.

Long-term debugging

The `print` statements described above are meant for quick debugging of one-time errors -- after figuring out the error, you would remove all the `print` statements.

However, sometimes we would like to leave the debugging code if we need to periodically test our file. It can get kind of annoying if every time we run our file, debugging messages pop up. One way to avoid this is to use a global `debug` variable:

```
debug = True

def foo(n):
i = 0
while i < n:
    i += func(i)
    if debug:
        print('DEBUG: i is', i)
```

Now, whenever we want to do some debugging, we can set the global `debug` variable to `True`, and when we don't want to see any debugging input, we can turn it to `False` (such a variable is called a "flag").

Interactive Debugging

One way a lot of programmers like to investigate their code is by use of an interactive REPL. That is, a terminal where you can directly run functions and inspect their outputs.

Typically, to accomplish this, you can run

```
python -i file.py
```

and one then has a session of python where all the definitions of `file.py` have already been executed.

If you are using the `ok` autograder, it has a specific tool that enables you to jump into the middle of a failing test case. Just run

```
python ok -q <question name> -i --local
```

and if you have a failing test case for that question, the setup code and doctest will be printed on the screen and run, and you will then have access to a terminal where you can execute commands related to the program.

PythonTutor Debugging

Sometimes the best way to understand what is going on with a given piece of python code is to create an environment diagram. While creating an environment diagram by hand can sometimes be tedious, the tool PythonTutor creates environment diagrams automatically. To use this tool, you can copy-paste your code into it, and run it. Alternatively, if you are working on an assignment using the `ok` autograder, you can run

```
python ok -q <question name> --trace --local
```

and a browser window should open up with your code.

Using `assert` statements

Python has a feature known as an `assert` statement, which lets you test that a condition is true, and print an error message otherwise in a single line. This is useful if you know that certain conditions need to be true at certain points in your program. For example, if you are writing a function that takes in an integer and doubles it, it might be useful to ensure that your input is in fact an integer. You can then write the following code

```
def double(x):
    assert isinstance(x, int), "The input to double(x) must be an integer"
    return 2 * x
```

Note that we aren't really debugging the `double` function here, what we're doing is ensuring that anyone who calls `double` is doing so with the right arguments. For example, if we have a function `g` that takes in a string and a number and adds the length of the string to twice the number, and it is implemented like so:

```
def g(x, y):
    return double(x) + y # should be double(y) + len(x)
```

Instead of getting a weird error about not being able to add a string and a number, we get a clean error that the argument to `double` must be an integer. This allows us to quickly narrow down the problem.

One *major* benefit of assert statements is that they are more than a debugging tool, you can leave them in code permanantly. A key principle in software development is that it is generally better for code to crash than produce an incorrect result, and having asserts in your code makes it far more likely that your code will crash if it has a bug in it.

Error Types

The following are common error types that Python programmers run into.

`SyntaxError`

- **Cause**: code syntax mistake
- **Example**:

```
  File "file name", line number
    def incorrect(f)
                    ^
SyntaxError: invalid syntax
```

- **Solution**: the `^` symbol points to the code that contains invalid syntax. The error message doesn't tell you *what* is wrong, but it does tell you *where*.
- **Notes**: Python will check for `SyntaxErrors` before executing any code. This is different from other errors, which are only raised during runtime.

`IndentationError`

- **Cause**: improper indentation
- **Example**:

```
  File "file name", line number
    print('improper indentation')
IndentationError: unindent does not match any outer indentation level
```

- **Solution**: The line that is improperly indented is displayed. Simply re-indent it.
- **Notes**: If you are inconsistent with tabs and spaces, Python will raise one of these. Make sure you use spaces! (It's just less of a headache in general in Python to use spaces and all cs61a content uses spaces).

`TypeError`

- **Cause 1**:
  - Invalid operand types for primitive operators. You are probably trying to add/subract/multiply/divide incompatible types.
  - **Example**:

```
TypeError: unsupported operand type(s) for +: 'function' and 'int'
```

- **Cause 2**:
  - Using non-function objects in function calls.
  - **Example**:

```
>>> square = 3
>>> square(3)
Traceback (most recent call last):
  ...
TypeError: 'int' object is not callable
```

- **Cause 3**:
  - Passing an incorrect number of arguments to a function.
  - **Example**:

```
>>> add(3)
Traceback (most recent call last):
  ...
TypeError: add expected 2 arguments, got 1
```

`NameError`

- **Cause**: variable not assigned to anything OR it doesn't exist. This includes function names.
- **Example**:

```
File "file name", line number
  y = x + 3
NameError: global name 'x' is not defined
```

- **Solution**: Make sure you are initializing the variable (i.e. assigning the variable to a value) before you use it.
- **Notes**: The reason the error message says "global name" is because Python will start searching for the variable from a function's local frame. If the variable is not found there, Python will keep searching the parent frames until it reaches the global frame. If it still can't find the variable, Python raises the error.

`IndexError`

- **Cause**: trying to index a sequence (e.g. a tuple, list, string) with a number that exceeds the size of the sequence.
- **Example**:

```
File "file name", line number
  x[100]
IndexError: tuple index out of range
```

- **Solution**: Make sure the index is within the bounds of the sequence. If you're using a variable as an index (e.g. `seq[x]`, make sure the variable is assigned to a proper index.

Common Bugs

Spelling

Python is *case sensitive*. The variable `hello` is not the same as `Hello` or `hello` or `helo`. This will usually show up as a `NameError`, but sometimes misspelled variables will actually have been defined. In that case, it can be difficult to find errors, and it is never gratifying to discover it's just a spelling mistake.

Missing Parentheses

A common bug is to leave off the closing parenthesis. This will show up as a `SyntaxError`. Consider the following code:

```
def fun():
    return foo(bar()   # missing a parenthesis here

fun()
```

Python will raise a `SyntaxError`, but will point to the line *after* the missing parenthesis:

```
File "file name", line "number"
    fun()
      ^
SyntaxError: invalid syntax
```

In general, if Python points a `SyntaxError` to a seemingly correct line, you are probably forgetting a parenthesis somewhere.

Missing close quotes

This is similar to the previous bug, but much easier to catch. Python will actually tell you the line that is missing the quote:

```
File "file name", line "number"
  return 'hi
           ^
SyntaxError: EOL while scanning string literal
```

`EOL` stands for "End of Line."

`=` vs. `==`

The single equal sign `=` is used for *assignment*; the double equal sign `==` is used for testing equivalence. The most common error of this form is something like:

```
if x = 3:
```

Infinite Loops

Infinite loops are often caused by `while` loops whose conditions never change. For example:

```
i = 0
while i < 10:
    print(i)
```

Sometimes you might have incremented the wrong counter:

```
i, n = 0, 0
while i < 10:
    print(i)
    n += 1
```

Off-by-one errors

Sometimes a `while` loop or recursive function might stop one iteration too short. Here, it's best to walk through the iteration/recursion to see what number the loop stops at.