DEBUGGING TIPS

COMPUTER SCIENCE 61A

1 Introduction

Every time a function is called, Python creates what is called a *stack frame* for that specific function to hold local variables and other information. Of course, functions will also make other function calls, which requires the creation of even more frames. So how are these frames organized?

Python uses what is called a *stack* to hold frames. Frames are stacked in the order that they are created. Every time you make a function call, you place a frame on the stack. You can only remove that frame when both of the following conditions are satisfied:

- All of the frames above it have been removed; in other words, the frame we want to remove is on the top of the stack.
- The function that the frame belongs to has terminated.

If the most recent function call raises an error (also known as an *exception*), Python will dump the stack – which you, the programmer, will see as a traceback message.

1.1 Traceback

The *traceback* message displays the state of the stack at the time the error was raised. Essentially, the traceback will show you the chain of function calls that caused your error. You can follow this chain to identify exactly which functions are the ones causing trouble.

For example, the following Python code will output a traceback message:

```
>>> def bug(f, x):
...     return f(x)
>>> bug(3, 4)
Traceback (most recent call last):
  File "<pyshell#29>", line 1, in <module>
    bug(3, 4)
```

```
  File "<pyshell#28>", line 2, in bug
    return f(x)
TypeError: 'int' object is not callable
```

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:

- **Function**: The name of the function with which the frame is associated.
- **Number**: The line number in the file where the function is defined. This saves you time trying to search through your code to find the buggy line.
- **File Name**: The name of the file where the function was written. This is especially helpful when writing large programs.

The second line in the pair, which is indented further in than the first line, 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 and in what context the function was being used, among other things.

Finally, notice that the traceback is organized with the "most recent call last". That means the line of code that is directly responsible for the error is at the *bottom* of the traceback message.

1.2 Error Statements

The very last line in a 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. 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.

1.3 Doctests

One other (important) thing: don't forget to run doctests! For almost every assignment (homework, projects), we will provide you with doctests for certain functions – they are there for your benefit. A quick run through the doctests will:

- Notify you of `SyntaxError`s
- Check for basic functionality of your assignment, though passing doctests does *not* guarantee your assignment is entirely correct.

To run a doctest, go to your terminal, `cd` to the directory in which your file is saved, and type:

```
python3 -m doctest <assignment>
```

This will run all the doctests contained in the file. Please use the doctests: it catches a lot of small, trivial errors (like improper indentation) that can nonetheless make it hard for your readers grade your assignments!

2 Common Errors

The following are common errors that Python programmers run into.

2.1 SyntaxError

- **Cause**: Code syntax mistake
- **Example**:

```
  File ``<File name>'', line <number>
    def incorrect(f)
                  ^
SyntaxError: invalid syntax
```

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

2.2 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**: Python will check for `IndentationError`s before executing any code.

2.3 TypeError

There are multiple possible causes:

- Invalid operand types for primitive operators. You are probably trying to add/subtract/multiply/divide incompatible types.

  Example:

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

  Solution: Change the operand that is incorrect. The order of the operand types shown will narrow down which operand is of an incorrect type. In the above example, the first operand to the + operator is incorrect.

- Using non-function objects in function calls.

  Example:

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

  Solution: This bug usually happens when you accidentally assign a variable to a non-function object instead of a function. Double-check the code where you assign that particular variable.

- Passing incorrect number of arguments to a function.

  Example:

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

  Solution: Make sure you pass in the correct number of arguments to the specified function; the error message tells you how many arguments there should be, and how many you tried to pass.

2.4 NameError

- **Cause**: Variable has not been assigned to anything, *or* it does not exist. This includes function names.
- **Example**:

```
  File <''file name''>, line <number>, in <function>
    y = x + 3
NameError: global name 'x' is not defined
```

- **Solution**: Make sure you are initializing the variable (i.e. assigning the variable a value / function object) before you use it.
- **Notes**: The reason the error message says "global name" is because Python will start searching for the specified variable from a function's local scope. If the variable is not found in the local scope, Python will keep searching outward until it reaches the global scope. If it still can't find a variable by that name in the global scope, Python will raise the error.

2.5 IndexError

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

```
  File <''file name''>, line <number>, in <function>
    s[100]
IndexError: tuple index out of range
```

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

3 Common Bugs

The following are not errors that Python will complain about, but rather mistakes that programmers unintentionally make that Python will not be able to catch.

3.1 Spelling and Capitalization

Remember that 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, the misspelled variable may actually have been defined. In that case, it can be difficult to find the error, and it is never gratifying to discover that it's just a spelling mistake.

3.2 Missing Parentheses

One of the most common bugs is to leave off a closing parenthesis (or a closing single quote, or a closing double quote, or a closing bracket, or so on). If you see something like this:

```
SyntaxError: EOL while scanning string literal
```

or anything with an `EOL` message, you most likely forgot a closing marker. Solution? Add it back in.

3.3 = 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:
```

3.4 Order of Operations

Remember that arithmetic is evaluated just like it would be in math. The following are listed from highest precedence to lowest:

- **Parentheses**: `()`
- **Exponents**: `**`
- **Multiplication**: `*`, **Division**: `/` or `//`, and **Modulo**: `%`
- **Addition**: `+`, and **Subtraction**: `-`

Boolean operators also have an order of precedence. As before, the list is organized from highest precedence to lowest:

- `not`: logical negation
- `and`: logical AND
- `or`: logical OR

This means that the following expression will return `True`.

```
False or not True and True or not False
```

which is equivalent to saying

```
False or ((not True) and True) or (not False)
```

3.5 Multiple Comparisons

Many beginners try to write "x and y are both greater than 0" as

```
x and y > 0
```

However, Python (and most other languages) require you to explicitly apply the comparison to each variable:

```
x > 0 and y > 0
```

3.6 Infinite Loops and Runaway Recursion

Infinite loops are often caused by `while` loops; you might have forgotten to increment or decrement the iterating variable. For example, the following will continually print 0, never stopping.

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

Poorly written recursion will cause a large traceback to occur, followed by

```
RuntimeError: Maximum recursion depth exceeded
```

If that occurs, you mostly likely forgot to write a base case, or your base case may need to be fixed.

3.7 Acknowledgments

This debugging guide was first written for CS61A, Summer 2012 by Albert Wu.