Debugging Help Sheet

Prepared by Ludovica Bindi and Dorukhan Yeşilli

This guide contains some basic information about debugging your code properly. It describes the general steps you should follow when debugging, the different kinds of errors that you may encounter and ways to fix them, and a (noncomprehensive) checklist that can help you when debugging. Finally, you can find some information about properly commenting on your code.

General steps to bug fixing

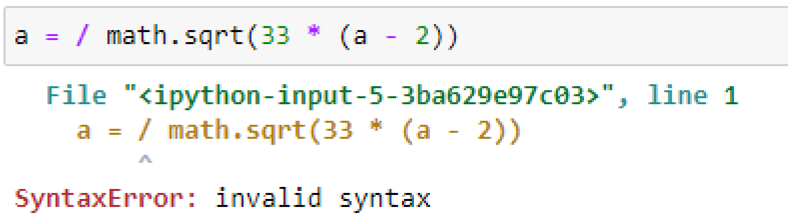
Here is the general process of dealing with bugs in your code:

1. **Observe the bug**. *"What makes me think there is a problem?"* Examples of bugs that you may have in this course are not being able to visualize what you want, not selecting the data you want with pandas, etc.
2. **Create a reproducible input.** *"How can I reliably reproduce this problem?"* Maybe the bug is due to a particular value in your input variables, sometimes you randomly see the bug and sometimes you don't. Try to understand when you see the bug by finding the (smallest) input values that raise the problem.
3. **Narrow the search space.** *"How can I narrow down to the code that could cause this problem?"* Reading the whole code you have to find the bug may be not feasible. Instead, start your search by looking at the most likely places, e.g. the last place where you made a change.
4. **Analyze.** *"What information can I gather from this identified code?"* Try to understand why the bug is there by reading the code. Check the official documentation if you are using libraries that are causing the problem.
    a. **Devise and run experiments.** *"How can I check my hypotheses about the issue?"* Run experiments (e.g., change the input values, not the code itself) with the code that you suspect is wrong. Iterate until you identify the root cause.
5. **Modify code to squash the bug.** *"How can I fix the issue, and confirm that the fix works?"* Don't start randomly changing the code, you should be able to explain the fix you make thanks to the analysis you carried out in the previous steps. And make sure that with the fix your code runs smoothly with all the problematic inputs that you found before.

Different kinds of errors

There are different kinds of errors that you can come across which require different solutions:

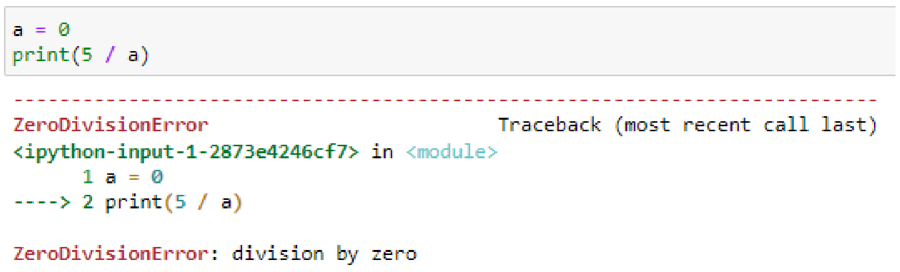
- **Syntax Error:**
    - It happens when you run a cell and as output, you see a message that starts with "SyntaxError". This means that the syntax of your code doesn't respect Python rules so Python is not able to run your code.



```
a = / math.sqrt(33 * (a - 2))
```

```
  File "<ipython-input-5-3ba629e97c03>", line 1
    a = / math.sqrt(33 * (a - 2))
        ^
SyntaxError: invalid syntax
```

        - The little “ ^ “ tells you where Python is finding the syntax problem, which is not necessarily where the error is. Sometimes the error is prior to the location of the error message, often on the preceding line.
    - Ways to fix the most common problems:
        - Check that you have a colon at the end of the header of every compound statement, including for, while, if, and def statements.
        - Check the indentation to make sure it lines up the way it is supposed to. Python can handle space and tabs, but if you mix them it can cause problems.
        - Make sure you are not using a Python keyword for a variable name (e.g., you can’t call a variable “if”). Here is a list of Python keywords.
        - Make sure that any strings in the code have matching quotation marks (e.g., “example’ is not correct, whereas “example” or ‘example’ are).
        - Every time that you have an open parenthesis (“(“, “[“, “{“), you have to have the matching closing parenthesis. And you can’t intertwine the parenthesis types: e.g., you can’t have “ (5 + [3 + 3) * 4] “ because after an open parathesis you can either have a new open parenthesis or the closing parenthesis of the same type. Plus, an unclosed opening operator—(, {, or [—makes Python continue with the next line as part of the current statement. Generally, an error occurs almost immediately in the next line.
        - Check for the classic = (assigning a value) instead of == (checking equality) inside a conditional.
- **Runtime errors:**
    - Runtime errors refer to errors that happen when you are executing your program. Your program is syntactically correct but it doesn’t work.
        - An example:



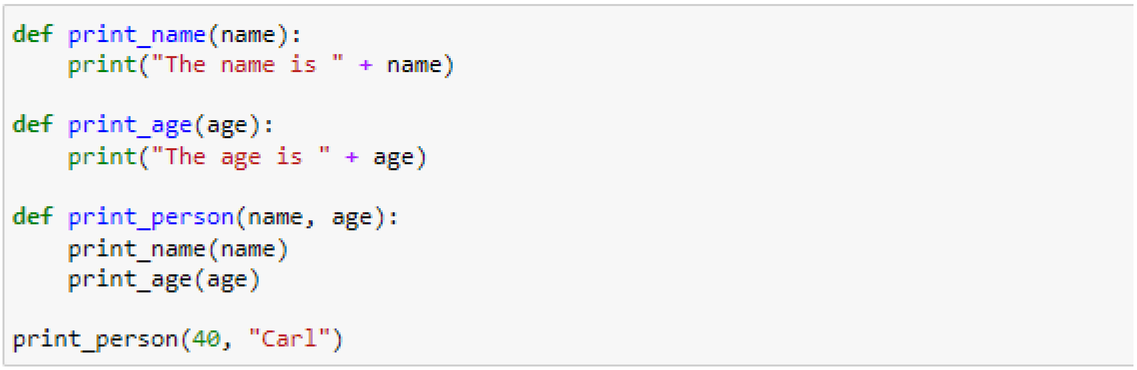
```
a = 0
print(5 / a)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-1-2873e4246cf7> in <module>
      1 a = 0
----> 2 print(5 / a)

ZeroDivisionError: division by zero
```

            - The program is syntactically correct but it creates an error because it is trying to divide a number by 0 which is not possible.
    - Here is a list of some of the most common runtime errors you may encounter and ways to fix themr:
        - *The program hangs*

    - You may have an infinite loop (that is the program keeps running infinitely because it is stuck in a for/while loop and it can’t get out of it).
    - Way to understand what is causing the problem: add a print statement at the end of the loop that prints the values of the variables in the condition and the value of the condition.
    - Fix: make sure that you are incrementing/changing the variables that determine the condition of the loop statement so that this loop can actually end; check that the condition is written as you want (e.g., check operators predecence)
- *You don’t know which functions are called*
    - You are not sure what the program is doing because there are many methods invocation in your code
    - Fix: If you are not sure how the flow of execution is moving through your program, add print statements to the beginning of each function with a message like “entering function foo,” where foo is the name of the function. Now when you run the program, it will print a trace of each function as it is invoked.
- *When you run the program you get an exception*
    - If something goes wrong during runtime, Python prints a message that includes the name of the problem detected (and a brief explanation/hint), the line of the program where the problem occurred, and a traceback, that is the function calls made in your code at a specific point.
    - The traceback identifies the function currently running that caused the problem, the function that invoked it, the function that invoked this last one, and so on. In other words, it traces the sequence of function invocations that got you to the problem. It also includes the line number in your file where each of these calls occurs.

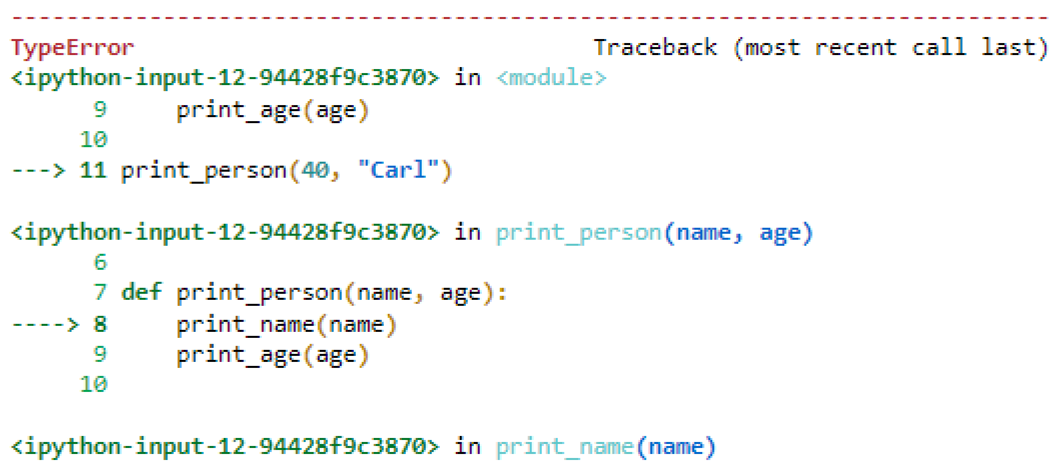


```python
def print_name(name):
    print("The name is " + name)

def print_age(age):
    print("The age is " + age)

def print_person(name, age):
    print_name(name)
    print_age(age)

print_person(40, "Carl")
```



```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-94428f9c3870> in <module>
      9     print_age(age)
     10 
---> 11 print_person(40, "Carl")

<ipython-input-12-94428f9c3870> in print_person(name, age)
      6 
      7 def print_person(name, age):
----> 8     print_name(name)
      9     print_age(age)
     10 

<ipython-input-12-94428f9c3870> in print_name(name)
```

            - You have to read the stacktrace from bottom to top: the line of code that causes the problem is the last one higlighited (line 2), in the last snippet of code (because we are directly printing an integer without making it a string first). In this case, this line belongs to a function. This function has been called at line 8 as highlighted by the second-to-last snippet of code. And so on.
            - The problem here is a “TypeError”.
        - Fix: The first step is to examine the place in the program where the error occurred and understand the error you get. Then see if you can figure out what happened there.
    - *The reason you have an error is an invalid input*
        - A very common reason why you may get a problem of this type could be that you used values that you were not supposed as inputs for other functions, expressions, etc.
        - Fix: ask yourself why you get such value, what the problematic values are, if you can check the value before using it and dealing with it if it is problematic, etc.
        - Special case: if sometimes you have the error and sometimes you don’t, it may be because you are using a random value somewhere and that only sometimes you are getting the value that generates the problem
            - Fix: a good place to start looking for the bug is where you use random values.

- **Semantic errors**
    - When your program is syntactically correct, runs without problems (no runtime errors) but you still don’t get what you want, the reason is that your code doesn’t do what you want it to do. In some ways, semantic errors are the hardest to debug, because Python doesn’t tell you what is going wrong, only you know what the program is supposed to do
    - In order to program, you need to have a mental model of what your program should look like, and the various steps you want to do in order to achieve the results. If you write a program that doesn’t do what you expect (but still has no runtime errors), very often the problem is not the code itself, it is what you have in mind that doesn’t work
    - The best way to correct your mental model is to break the program into its main parts (usually the functions and methods) and test each part independently. Once you find the discrepancy between your model and reality, you can solve the problem.
    - There is no single way of fixing this. Here are some suggestions:
        - Divide functions into smaller ones so that a single function does one thing and debug each single function
        - Break a complex expression into a series of assignments to temporary variables (use temporary variable).
            - Another problem that can occur with big expressions is that the order of evaluation may not be what you expect. E.g.,

multiplication and divisions have the same precedence and are executed from left to right
        - Make sure there is a return statement when you need one
        - Make sure that in the functions you use only the elements that are passed in the argument of the function to keep the components separated
- **Problems due to Jupyter Notebook**
    - If everything seems correct but you still get the wrong result e.g. after fixing a bug, the problem could be that the variables that you have created so far contain values that are not the ones you want/are not what they are supposed to be. This is because Python is a scripting language so variables don't "die" when you run a cell in the notebook but keep existing as long as the kernel is alive.
        - Fix: restart the kernel and re-run the whole thing
    - If you just downloaded a library and you still can't import it on your notebook, restart the kernel so that Python can updates its libraries
- **General comments**
    - Don't forget to remove/comment the print statements when you are done with debugging!
    - Do read the error message and do read the stack trace! They hint at what the error is and where it is
    - How to properly use Jupyter Notebook:
        - Make sure that the cells are in the order they need to be run too (e.g., having cell 5 containig the initialization of the variables and cell 4 containing the code that deals with that variables) so that you can easily use the notebook even days later after you worked on it (you don't have to remember weird orders of cells execution)
        - Split the code into different cells so that you can easily look at what is happening at each step

Basic debugging checklist

Here is a basic checklist that can help you when debugging:

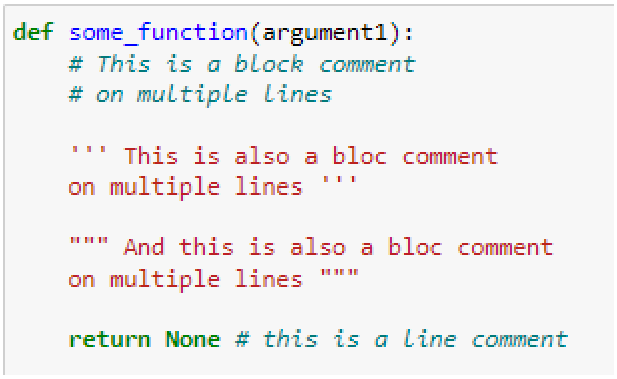
- *Data Initialization*
    - Did you initialize the variable if needed?
    - Any improperly initialized variables?
    - Any misspelled variable names?
    - Variables with similar names confused?
- *Computation*
    - When slicing arrays, strings, etc. did you use the correct bounds?
    - Any non-integer indexing?
    - Did you save the results you computed and that you wanted to use somewhere else?
    - Division by zero?
    - Operator precedence understood?
- *Comparisons*
    - Comparison relationships correct?
    - = used instead of ==?

    - Boolean expressions correct?
    - Operator precedence understood?
- *Control Flow*
    - All cases covered in if or case statements?
    - Should there be a default case in if..else statements?
    - If statement used when elif should be used or vice versa?
    - Will each loop terminate?
        - Is the loop condition achievable?
        - Do you increment / modify the variables of the condition so that the condition can become false?
        - Are the variables of the loop condition properly initialized?
    - Any loop bypasses because of entry conditions?
    - Indentation correct? Remember indentation is essential for Python syntax.
- *Method Calls (especially tricky when using libraries)*
    - Method names spelled correctly?
    - Method names capitalized correctly?
    - Number of arguments correct?
    - Types of arguments correct?
    - Order of the arguments correct?
    - Name of arguments correct?
    - Package containing method used imported correctly?
    - Did you only use the arguments or called variables instantiated outside the function?
- *Input/Output*
    - Right type of file for the method you are using (e.g., xlsx won't be opened with pandas.read_csv)?
- *Other Checks*
    - Check eventual warning message
    - Import statements correct?
    - Do you have all the libraries needed? With their right version?

Comments on your code

Writing good comments on your code ensure the understandability of your code and can help you write good code and be more efficient at debugging.

- There are two kinds of comments:
    - Block comments: one or more sentences. They apply to the code that directly follows them and have the same indentation with it
        - Two ways to make them: using a “#” at the beginning of each line of code or putting the texts between two “''' “ or “ """“
    - Inline comment: on the same line as the piece of code it comments. It should be shorter and should be used with caution since they tend to be visually mixed with the lines of code
        - Done with a “#” before the comment



```python
def some_function(argument1):
    # This is a block comment
    # on multiple lines

    ''' This is also a bloc comment
    on multiple lines '''

    """ And this is also a bloc comment
    on multiple lines """

    return None # this is a line comment
```

- Some basic guidelines to write meaningful comments:
    - Use block comments for general comments about your code and use the line comments to describe something specific of the line that you are commenting
    - If you use self-explanatory names for variables and functions, fewer comments are needed to make the code clear
    - Use comments to section the code into several logical parts making it easier to navigate. Describe each part with some block comments
    - Avoid obvious code comments
    - Code comments should be as precise and laconic as possible, giving all the necessary details about their piece of code and excluding any irrelevant information such as observations from the previous code, intentions for the future, or parenthesis.
    - Also, since the code usually implies some dynamics (it does, creates, removes, etc.), a good practice is to use verbs rather than nouns
    - When commenting on a function, add right after the function name a general description of the function, the arguments that are needed and their type, the return values and their type

Resources

https://en.wikibooks.org/wiki/Think_Python/Debugging
https://web.stanford.edu/class/archive/cs/cs107/cs107.1226/resources/debugging.html
http://facweb.cs.depaul.edu/sjost/it212/documents/debug-checklist.htm
https://towardsdatascience.com/jupyter-notebooks-avoid-making-these-4-mistakes-e5cbad1d473d
https://towardsdatascience.com/the-art-of-writing-efficient-code-comments-692213ed71b1