Debugging

Different kinds of errors can occur in a program, and it is useful to distinguish among them in order to track them down more quickly:

1. Syntax errors are produced by Python when it is translating the source code into byte code. They usually indicate that there is something wrong with the syntax of the program. Example: Omitting the colon at the end of a `def` statement yields the somewhat redundant message `SyntaxError: invalid syntax`.
2. Runtime errors are produced by the runtime system if something goes wrong while the program is running. Most runtime error messages include information about where the error occurred and what functions were executing. Example: An infinite recursion eventually causes a runtime error of maximum recursion depth exceeded.
3. Semantic errors are problems with a program that compiles and runs but doesn't do the right thing. Example: An expression may not be evaluated in the order you expect, yielding an unexpected result.

The first step in debugging is to figure out which kind of error you are dealing with. Although the following sections are organized by error type, some techniques are applicable in more than one situation.

Syntax errors

Syntax errors are usually easy to fix once you figure out what they are. Unfortunately, the error messages are often not helpful. The most common messages are `SyntaxError: invalid syntax` and `SyntaxError: invalid token`, neither of which is very informative.

On the other hand, the message does tell you where in the program the problem occurred. Actually, it tells you where Python noticed a 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.

If you are building the program incrementally, you should have a good idea about where the error is. It will be in the last line you added.

If you are copying code from a book, start by comparing your code to the book's code very carefully. Check every character. At the same time, remember that the book might be wrong, so if you see something that looks like a syntax error, it might be.

Here are some ways to avoid the most common syntax errors:

1. Make sure you are not using a Python keyword for a variable name.
2. Check that you have a colon at the end of the header of every compound statement, including `for`, `while`, `if`, and `def` statements.
3. Check that indentation is consistent. You may indent with either spaces or tabs but it's best not to mix them. Each level should be nested the same amount.
4. Make sure that any strings in the code have matching quotation marks.
5. If you have multiline strings with triple quotes (single or double), make sure you have terminated the string properly. An unterminated string may cause an `invalid token` error at the end of your program, or it may treat the following part of the program as a string until it comes to the next string. In the second case, it might not produce an error message at all!
6. An unclosed bracket — (, {, or [— makes Python continue with the next line as part of the current statement. Generally, an error occurs almost immediately in the next line.
7. Check for the classic `=` instead of `==` inside a conditional.

If nothing works, move on to the next section...

I can't get my program to run no matter what I do.

If the compiler says there is an error and you don't see it, that might be because you and the compiler are not looking at the same code. Check your programming environment to make sure that the program you are editing is the one Python is trying to run. If you are not sure, try putting an obvious and deliberate syntax error at the beginning of the program. Now run (or import) it again. If the compiler doesn't find the new error, there is probably something wrong with the way your environment is set up.

If this happens, one approach is to start again with a new program like Hello, World!, and make sure you can get a known program to run. Then gradually add the pieces of the new program to the working one.

Runtime errors

Once your program is syntactically correct, Python can import it and at least start running it. What could possibly go wrong?

My program does absolutely nothing.

This problem is most common when your file consists of functions and classes but does not actually invoke anything to start execution. This may be intentional if you only plan to import this module to supply classes and functions.

If it is not intentional, make sure that you are invoking a function to start execution, or execute one from the interactive prompt. Also see the Flow of Execution section below.

My program hangs.

If a program stops and seems to be doing nothing, we say it is hanging. Often that means that it is caught in an infinite loop or an infinite recursion.

1. If there is a particular loop that you suspect is the problem, add a `print` statement immediately before the loop that says entering the loop and another immediately after that says exiting the loop.
2. Run the program. If you get the first message and not the second, you've got an infinite loop. Go to the Infinite Loop section below.
3. Most of the time, an infinite recursion will cause the program to run for a while and then produce a RuntimeError: Maximum recursion depth exceeded error. If that happens, go to the Infinite Recursion section below.
4. If you are not getting this error but you suspect there is a problem with a recursive method or function, you can still use the techniques in the Infinite Recursion section.
5. If neither of those steps works, start testing other loops and other recursive functions and methods.
6. If that doesn't work, then it is possible that you don't understand the flow of execution in your program. Go to the Flow of Execution section below.

Infinite Loop

If you think you have an infinite loop and you think you know what loop 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.

For example:

```
while x > 0 and y < 0:
    # Do something to x
    # Do something to y

    print("x: ", x)
    print("y: ", y)
    print("condition: ", (x > 0 and y < 0))
```

Now when you run the program, you will see three lines of output for each time through the loop. The last time through the loop, the condition should be `False`. If the loop keeps going, you will be able to see the values of `x` and `y`, and you might figure out why they are not being updated correctly.

In a development environment like PyScripter, one can also set a breakpoint at the start of the loop, and single-step through the loop. While you do this, inspect the values of `x` and `y` by hovering your cursor over them.

Of course, all programming and debugging require that you have a good mental model of what the algorithm ought to be doing: if you don't understand what ought to happen to `x` and `y`, printing or inspecting its value is of little use. Probably the best place to debug the code is away from your computer, working on your understanding of what should be happening.

Infinite Recursion

Most of the time, an infinite recursion will cause the program to run for a while and then produce a `Maximum recursion depth exceeded` error.

If you suspect that a function or method is causing an infinite recursion, start by checking to make sure that there is a base case. In other words, there should be some condition that will cause the function or method to return without making a recursive invocation. If not, then you need to rethink the algorithm and identify a base case.

If there is a base case but the program doesn't seem to be reaching it, add a `print` statement at the beginning of the function or method that prints the parameters. Now when you run the program, you will see a few lines of output every time the function or method is invoked, and you will see the parameters. If the parameters are not moving toward the base case, you will get some ideas about why not.

Once again, if you have an environment that supports easy single-stepping, breakpoints, and inspection, learn to use them well. It is our opinion that walking through code step-by-step builds the best and most accurate mental model of how computation happens. Use it if you have it!

Flow of Execution

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.

If you're not sure, step through the program with your debugger.

When I run the program I get an exception.

If something goes wrong during runtime, Python prints a message that includes the name of the exception, the line of the program where the problem occurred, and a traceback.

Put a breakpoint on the line causing the exception, and look around!

The traceback identifies the function that is currently running, and then the function that invoked it, and then the function that invoked *that*, and so on. In other words, it traces the path of function invocations that got you to where you are. It also includes the line number in your file where each of these calls occurs.

The first step is to examine the place in the program where the error occurred and see if you can figure out what happened. These are some of the most common runtime errors:

NameError
: You are trying to use a variable that doesn't exist in the current environment. Remember that local variables are local. You cannot refer to them from outside the function where they are defined.

TypeError
: There are several possible causes:

1. You are trying to use a value improperly. Example: indexing a string, list, or tuple with something other than an integer.

2. There is a mismatch between the items in a format string and the items passed for conversion. This can happen if either the number of items does not match or an invalid conversion is called for.
3. You are passing the wrong number of arguments to a function or method. For methods, look at the method definition and check that the first parameter is `self`. Then look at the method invocation; make sure you are invoking the method on an object with the right type and providing the other arguments correctly.

KeyError

You are trying to access an element of a dictionary using a key value that the dictionary does not contain.

AttributeError

You are trying to access an attribute or method that does not exist.

IndexError

The index you are using to access a list, string, or tuple is greater than its length minus one. Immediately before the site of the error, add a `print` statement to display the value of the index and the length of the sequence. Is the sequence the right size? Is the index the right value?

I added so many `print` statements I get inundated with output.

One of the problems with using `print` statements for debugging is that you can end up buried in output. There are two ways to proceed: simplify the output or simplify the program.

To simplify the output, you can remove or comment out `print` statements that aren't helping, or combine them, or format the output so it is easier to understand.

To simplify the program, there are several things you can do. First, scale down the problem the program is working on. For example, if you are sorting a sequence, sort a *small* sequence. If the program takes input from the user, give it the simplest input that causes the problem.

Second, clean up the program. Remove dead code and reorganize the program to make it as easy to read as possible. For example, if you suspect that the problem is in a deeply nested part of the program, try rewriting that part with simpler structure. If you suspect a large function, try splitting it into smaller functions and testing them separately.

Often the process of finding the minimal test case leads you to the bug. If you find that a program works in one situation but not in another, that gives you a clue about what is going on.

Similarly, rewriting a piece of code can help you find subtle bugs. If you make a change that you think doesn't affect the program, and it does, that can tip you off.

You can also wrap your debugging print statements in some condition, so that you suppress much of the output. For example, if you are trying to find an element using a binary search, and it is not working, you might code up a debugging print statement inside a conditional: if the range of candidate elements is less that 6, then print debugging information, otherwise don't print.

Similarly, breakpoints can be made conditional: you can set a breakpoint on a statement, then edit the breakpoint to say "only break if this expression becomes true".

Semantic errors

In some ways, semantic errors are the hardest to debug, because the compiler and the runtime system provide no information about what is wrong. Only you know what the program is supposed to do, and only you know that it isn't doing it.

The first step is to make a connection between the program text and the behavior you are seeing. You need a hypothesis about what the program is actually doing. One of the things that makes that hard is that computers run so fast.

You will often wish that you could slow the program down to human speed, and with some debuggers you can. But the time it takes to insert a few well-placed `print` statements is often short compared to

setting up the debugger, inserting and removing breakpoints, and walking the program to where the error is occurring.

My program doesn't work.

You should ask yourself these questions:

1. Is there something the program was supposed to do but which doesn't seem to be happening? Find the section of the code that performs that function and make sure it is executing when you think it should.
2. Is something happening that shouldn't? Find code in your program that performs that function and see if it is executing when it shouldn't.
3. Is a section of code producing an effect that is not what you expected? Make sure that you understand the code in question, especially if it involves invocations to functions or methods in other Python modules. Read the documentation for the functions you invoke. Try them out by writing simple test cases and checking the results.

In order to program, you need to have a mental model of how programs work. If you write a program that doesn't do what you expect, very often the problem is not in the program; it's in your mental model.

The best way to correct your mental model is to break the program into its components (usually the functions and methods) and test each component independently. Once you find the discrepancy between your model and reality, you can solve the problem.

Of course, you should be building and testing components as you develop the program. If you encounter a problem, there should be only a small amount of new code that is not known to be correct.

I've got a big hairy expression and it doesn't do what I expect.

Writing complex expressions is fine as long as they are readable, but they can be hard to debug. It is often a good idea to break a complex expression into a series of assignments to temporary variables.

For example:

```
self.hands[i].add_card(self.hands[self.find_neighbor(i)].pop_card())
```

This can be rewritten as:

```
neighbor = self.find_neighbor (i)
picked_card = self.hands[neighbor].pop_card()
self.hands[i].add_card(picked_card)
```

The explicit version is easier to read because the variable names provide additional documentation, and it is easier to debug because you can check the types of the intermediate variables and display or inspect their values.

Another problem that can occur with big expressions is that the order of evaluation may not be what you expect. For example, if you are translating the expression `x/2pi` into Python, you might write:

```
y = x / 2 * math.pi
```

That is not correct because multiplication and division have the same precedence and are evaluated from left to right. So this expression computes `(x/2)pi`.

A good way to debug expressions is to add parentheses to make the order of evaluation explicit:

```
y = x / (2 * math.pi)
```

Whenever you are not sure of the order of evaluation, use parentheses. Not only will the program be correct (in the sense of doing what you intended), it will also be more readable for other people who haven't memorized the rules of precedence.

I've got a function or method that doesn't return what I expect.

If you have a `return` statement with a complex expression, you don't have a chance to print the `return` value before returning. Again, you can use a temporary variable. For example, instead of:

```
return self.hands[i].remove_matches()
```

you could write:

```
count = self.hands[i].remove_matches()
return count
```

Now you have the opportunity to display or inspect the value of `count` before returning.

I'm really, really stuck and I need help.

First, try getting away from the computer for a few minutes. Computers emit waves that affect the brain, causing these effects:

1. Frustration and/or rage.
2. Superstitious beliefs (the computer hates me) and magical thinking (the program only works when I wear my hat backward).
3. Random-walk programming (the attempt to program by writing every possible program and choosing the one that does the right thing).

If you find yourself suffering from any of these symptoms, get up and go for a walk. When you are calm, think about the program. What is it doing? What are some possible causes of that behavior? When was the last time you had a working program, and what did you do next?

Sometimes it just takes time to find a bug. We often find bugs when we are away from the computer and let our minds wander. Some of the best places to find bugs are trains, showers, and in bed, just before you fall asleep.

No, I really need help.

It happens. Even the best programmers occasionally get stuck. Sometimes you work on a program so long that you can't see the error. A fresh pair of eyes is just the thing.

Before you bring someone else in, make sure you have exhausted the techniques described here. Your program should be as simple as possible, and you should be working on the smallest input that causes the error. You should have `print` statements in the appropriate places (and the output they produce should be comprehensible). You should understand the problem well enough to describe it concisely.

When you bring someone in to help, be sure to give them the information they need:

1. If there is an error message, what is it and what part of the program does it indicate?
2. What was the last thing you did before this error occurred? What were the last lines of code that you wrote, or what is the new test case that fails?
3. What have you tried so far, and what have you learned?

Good instructors and helpers will also do something that should not offend you: they won't believe when you tell them *"I'm sure all the input routines are working just fine, and that I've set up the data correctly!"*. They will want to validate and check things for themselves. After all, your program has a bug. Your understanding and inspection of the code have not found it yet. So you should expect to have your assumptions challenged. And as you gain skills and help others, you'll need to do the same for them.

When you find the bug, take a second to think about what you could have done to find it faster. Next time you see something similar, you will be able to find the bug more quickly.

Remember, the goal is not just to make the program work. The goal is to learn how to make the program work.