What is Debugging?

Programming is a complex process. Since it is done by human beings, errors may often occur. Programming errors are called **bugs** and the process of tracking them down and correcting them is called **debugging**. Some claim that in 1945, a dead moth caused a problem on relay number 70, panel F, of one of the first computers at Harvard, and the term **bug** has remained in use since. For more about this historic event, see first bug (http://en.wikipedia.org/wiki/File:H96566k.jpg).

Three kinds of errors can occur in a program: syntax errors (http://en.wikipedia.org/wiki/Syntax_error), runtime errors (http://en.wikipedia.org/wiki/Runtime_error), and semantic errors (http://en.wikipedia.org/wiki/Logic_error). It is useful to distinguish between them in order to track them down more quickly.

Check your understanding

intr-9: Debugging is:

- (•) a) tracking down programming errors and correcting them.
- ( ) b) removing all the bugs from your house.
- ( ) c) finding all the bugs in the program.
- ( ) d) fixing the bugs in the program.

Check Me | Compare Me

Correct!! Programming errors are called bugs and the process of finding and removing them from a program is called debugging.

Syntax errors

Python can only execute a program if the program is syntactically correct; otherwise, the process fails and returns an error message. **Syntax** refers to the structure of a program and the rules about that structure. For example, in English, a sentence must begin with a capital letter and end with a period. this sentence contains a **syntax error**. So does this one

For most readers, a few syntax errors are not a significant problem, which is why we can read the poetry of e. e. cummings without problems. Python is not so forgiving. If there is a single syntax error anywhere in your program, Python will display an error message and quit. You will not be able to complete the execution your

program. During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. However, as you gain experience, you will make fewer errors and you will also be able to find your errors faster.

Check your understanding

intr-10: Which of the following is a syntax error?

- a) Attempting to divide by 0.
- b) Forgetting a colon at the end of a statement where one is required.
- c) Forgetting to divide by 100 when printing a percentage amount.

Check Me | Compare Me

Correct!! This is a problem with the formal structure of the program. Python knows where colons are required and can detect when one is missing simply by looking at the code without running it.

intr-11: Who or what typically finds syntax errors?

- a) The programmer.
- b) The compiler / interpreter.
- c) The computer.
- d) The teacher / instructor.

Check Me | Compare Me

Correct!! The compiler and / or interpreter is a computer program that determines if your program is written in a way that can be translated into machine language for execution.

Runtime Errors

The second type of error is a runtime error, so called because the error does not appear until you run the program. These errors are also called **exceptions** because they usually indicate that something exceptional (and bad) has happened.

Runtime errors are rare in the simple programs you will see in the first few chapters, so it might be a while before you encounter one.

Check your understanding

intr-12: Which of the following is a run-time error?

- (•) a) Attempting to divide by 0.
- ( ) b) Forgetting a colon at the end of a statement where one is required.
- ( ) c) Forgetting to divide by 100 when printing a percentage amount.

Check Me | Compare Me

Correct!! Python cannot reliably tell if you are trying to divide by 0 until it is executing your program (e.g., you might be asking the user for a value and then dividing by that value—you cannot know what value the user will enter before you run the program).

Semantic Errors

The third type of error is the **semantic error**. If there is a semantic error in your program, it will run successfully in the sense that the computer will not generate any error messages. However, your program will not do the right thing. It will do something else. Specifically, it will do what you told it to do.

The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

Check your understanding

intr-13: Which of the following is a semantic error?

- ( ) a) Attempting to divide by 0.
- ( ) b) Forgetting a semi-colon at the end of a statement where one is required.
- (•) c) Forgetting to divide by 100 when printing a percentage amount.

Check Me | Compare Me

Correct!! This will produce the wrong answer because the programmer implemented the solution incorrectly. This is a semantic error.

Experimental Debugging

One of the most important skills you will acquire is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.

In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see.

Debugging is also like an experimental science. Once you have an idea what is going wrong, you modify your program and try again. If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one. As Sherlock Holmes pointed out, When you have eliminated the impossible, whatever remains, however improbable, must be the truth. (A. Conan Doyle, *The Sign of Four*)

For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should start with a program that does *something* and make small modifications, debugging them as you go, so that you always have a working program.

For example, Linux is an operating system kernel that contains millions of lines of code, but it started out as a simple program Linus Torvalds used to explore the Intel 80386 chip. According to Larry Greenfield, one of Linus's earlier projects was a program that would switch between displaying AAAA and BBBB. This later evolved to Linux (*The Linux Users' Guide* Beta Version 1).

Later chapters will make more suggestions about debugging and other programming practices.

Check your understanding

intr-14: The difference between programming and debugging is:

a) programming is the process of writing and gradually debugging a program until it does what you want.

b) programming is creative and debugging is routine.

c) programming is fun and debugging is work.

d) there is no difference between them.

Check Me | Compare Me

Correct!! Programming is the writing of the source code and debugging is the process of finding and correcting all the errors within the program until it is correct.

Formal and Natural Languages

Natural languages are the languages that people speak, such as English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved naturally.

Formal languages are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And most importantly:

> *Programming languages are formal languages that have been designed to express computations.*

Formal languages tend to have strict rules about syntax. For example, `3+3=6` is a syntactically correct mathematical statement, but `3=+6$` is not. H_2O is a syntactically correct chemical name, but $_2Zz$ is not.

Syntax rules come in two flavors, pertaining to **tokens** and structure. Tokens are the basic elements of the language, such as words, numbers, and chemical elements. One of the problems with `3=+6$` is that `$` is not a legal token in mathematics (at least as far as we know). Similarly, $_2Zz$ is not legal because there is no element with the abbreviation `Zz`.

The second type of syntax rule pertains to the **structure** of a statement— that is, the way the tokens are arranged. The statement `3=+6$` is structurally illegal because you can't place a plus sign immediately after an equal sign. Similarly, molecular formulas have to have subscripts after the element name, not before.

When you read a sentence in English or a statement in a formal language, you have to figure out what the structure of the sentence is (although in a natural language you do this subconsciously). This process is called **parsing**.

For example, when you hear the sentence, "The other shoe fell", you understand that the other shoe is the subject and fell is the verb. Once you have parsed a sentence, you can figure out what it means, or the **semantics** of the sentence. Assuming that you know what a shoe is and what it means to fall, you will understand the general implication of this sentence.

Although formal and natural languages have many features in common — tokens, structure, syntax, and semantics — there are many differences:

ambiguity

Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of context.

redundancy

In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of

redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.

literalness

Formal languages mean exactly what they say. On the other hand, natural languages are full of idiom and metaphor. If someone says, “The other shoe fell”, there is probably no shoe and nothing falling.

> **Tip**
>
> You’ll need to find the original joke to understand the idiomatic meaning of the other shoe falling. *Yahoo! Answers* thinks it knows!

People who grow up speaking a natural language—everyone—often have a hard time adjusting to formal languages. In some ways, the difference between formal and natural language is like the difference between poetry and prose, but more so:

poetry

Words are used for their sounds as well as for their meaning, and the whole poem together creates an effect or emotional response. Ambiguity is not only common but often deliberate.

prose

The literal meaning of words is more important, and the structure contributes more meaning. Prose is more amenable to analysis than poetry but still often ambiguous.

program

The meaning of a computer program is unambiguous and literal, and can be understood entirely by analysis of the tokens and structure.

Here are some suggestions for reading programs (and other formal languages). First, remember that formal languages are much more dense than natural languages, so it takes longer to read them. Also, the structure is very important, so it is usually not a good idea to read from top to bottom, left to right. Instead, learn to parse the program in your head, identifying the tokens and interpreting the structure. Finally, the details matter. Little things like spelling errors and bad punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

Check your understanding

intr-15: The differences between natural and formal languages include:

- a) natural languages can be parsed while formal languages cannot.
- b) ambiguity, redundancy, and literalness.
- c) there are no differences between natural and formal languages.
- d) tokens, structure, syntax, and semantics.

Check Me | Compare Me

Correct!! All of these can be present in natural languages, but cannot exist in formal languages.

intr-16: True or False: Reading a program is like reading other kinds of text.

a) True

b) False

Check Me | Compare Me

Correct!! It usually takes longer to read a program because the structure is as important as the content and must be interpreted in smaller pieces for understanding.

A Typical First Program

Traditionally, the first program written in a new language is called *Hello, World!* because all it does is display the words, Hello, World! In Python, the source code looks like this.

```
print("Hello, World!")
```

This is an example of using the **print function**, which doesn't actually print anything on paper. It displays a value on the screen. In this case, the result is the phrase:

```
Hello, World!
```

Here is the example in activecode. Give it a try!

```
1 print("Hello, World!")
2
```

ActiveCode: 1 (ch01_2)

Run

```
Hello, World!
```

The quotation marks in the program mark the beginning and end of the value. They don't appear in the result.

Some people judge the quality of a programming language by the simplicity of the Hello, World! program. By this standard, Python does about as well as possible.

Check your understanding

intr-17: The print function:

- a) sends information to the printer to be printed on paper.
- b) displays a value on the screen.
- c) tells the computer to put the information in print, rather than cursive, format.
- d) tells the computer to speak the information.

Check Me | Compare Me

Correct!! Yes, the print function is used to display the value of the thing being printed.

Comments

As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why. For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are called comments.

A **comment** in a computer program is text that is intended only for the human reader - it is completely ignored by the interpreter. In Python, the # token starts a comment. The rest of the line is ignored. Here is a new version of *Hello, World!*.

```
```

```
#-----------------------------------------------
# This demo program shows off how elegant Python is!
# Written by Joe Soap, December 2010.
# Anyone may freely copy or modify this program.
#-----------------------------------------------

print("Hello, World!")     # Isn't this easy!

```

ActiveCode: 2 (ch01_3)

Run

```
Hello, World!
```

Notice that when you run this program, it still only prints the phrase Hello, World! None of the comments appear. You'll also notice that we've left a blank line in the program. Blank lines are also ignored by the interpreter, but comments and blank lines can make your programs much easier for humans to parse. Use them liberally!

Check your understanding

intr-18: What are comments for?

- a) To tell the computer what you mean in your program.
- b) For the people who are reading your code to know, in natural language, what the program is doing.
- c) Nothing, they are extraneous information that is not needed.
- d) Nothing in a short program. They are only needed for really large programs.

Check Me | Compare Me

Correct!! The computer ignores comments. It's for the humans that will "consume" your program.

Note

This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

```
1 print ("Hello, world.")
2
```

ActiveCode: 3 (scratch_01)

Run

```
Hello, world.
```